AFRL-IF-RS-TR-2005-322
Final Technical Report
September 2005

EFFECTS BASED OPERATIONS (EBO)

University of Massachusetts

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

STINFO FINAL REPORT

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2005-322 has been reviewed and is approved for publication

APPROVED: /s/

DUANE GILMOUR
Project Engineer

FOR THE DIRECTOR: /s/

JAMES A. COLLINS, Deputy Chief
Advanced Computing Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	SEPTEMBER 2005	Final Sep 01 – Aug 04

4. TITLE AND SUBTITLE
EFFECTS BASED OPERATIONS (EBO)

5. FUNDING NUMBERS
C - F30602-01-1-0589
PE - 62702F
PR - 558B
TA - II
WU - 03

6. AUTHOR(S)
Paul R. Cohen

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of Massachusetts
Computer Science Department
Experimental Knowledge Systems Laboratory
Amherst Massachusetts 01003

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Air Force Research Laboratory/IFTC
26 Electronic Parkway
Rome New York 13441-4514

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
AFRL-IF-RS-TR-2005-322

11. SUPPLEMENTARY NOTES

AFRL Project Engineer: Duane Gilmour/IFTC/(315) 330-3550/ Duane.Gilmour@rl.af.mil

12a. DISTRIBUTION / AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 Words)*

Several parts of an integrated decision aid for Effects Based Operations (EBO) were designed and implemented. These extensive improvements to the Capture the Flag (CtF) wargaming system included a new declarative executable action specification language (Tapir) and simulation definition language (Krill); improved models of morale and defeat (Defeat Mechanisms); and a semi-graphical Action Model language. Combined, these pieces help to create models that can be used to evaluate Courses of Action (COAs) for EBO and other military operations. Effects based reasoning requires models of effects. These must distinguish physical from behavioral effects and physical structures from mental ones. Many effects are hidden and cannot be measured directly, especially effects on mental structures and behaviors. While physical effects, such as dropping a bridge can be confirmed, effects on morale, courage, and other intentional states can only be inferred from observable indicators, and will generally be uncertain. The original CtF system modeled only physical force on force conflict. The new Action Models and Defeat Mechanisms provide CtF with a much broader range of measurable effects. Krill and Tapir make it possible to describe plans and actions declaratively, while also supporting Monte Carlo simulation for the measurement of uncertain, probabilistic and dynamical events.

14. SUBJECT TERMS
Effects Based Operations, EBO, Monte Carlo Simulation, Wargaming, Capture The Flag, Course Of Action, COA

15. NUMBER OF PAGES
58

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Table of Contents

List of Figures

1. Introduction

The University of Massachusetts Experimental Knowledge Systems Laboratory (EKSL) designed and implemented several parts of an integrated decision aid for Effects Based Operations (EBO). These included extensive improvements to its Capture the Flag (CtF) wargaming system, a new declarative executable action specification language (Tapir) and simulation definition language (Krill), improved models of morale and defeat (Defeat Mechanisms), and a semi-graphical Action Model language. Combined, these pieces help to create models that can be used to evaluate Courses of Action (COAs) for EBO and other military operations.

Effects based reasoning requires models of effects. These must distinguish physical from behavioral effects and physical structures from mental ones. Many effects are *hidden* and cannot be measured directly, especially effects on mental structures and behaviors. While physical effects, such as dropping a bridge can be confirmed, effects on morale, courage, and other *intentional* states can only be inferred from observable *indicators*, and will generally be uncertain. The original Capture the Flag system modeled only physical force on force conflict. The new Action Models and Defeat Mechanisms provide CtF with a much broader range of measurable effects. Krill and Tapir make it possible to describe plans and actions declaratively, while also supporting Monte Carlo simulation for the measurement of uncertain, probabilistic and dynamical events.

2.0 Extensions to CtF: Steps towards a Component Model

At the start of EKSL's EBO work, the Capture the Flag war gaming system was a mature force on force simulation of military tactics (see Appendix 2 for a description). Although we had begun investigating Defeat Mechanisms, these were not part of CtF nor could they be used for planning. Furthermore, actions in CtF were implemented as code. They could not be understood, except by programmers, and they provided no semantic information beyond their names. That is, there was nothing about an *attack* action that indicated it would damage enemy units whereas a *move* action would not. These opaque actions could not function as model building components. Like puzzle pieces that all looked the same, it was impossible to tell which actions could go with other actions to build a coherent plan. Nor was it possible to tell what an action was supposed to do or how to grade the actual or supposed effects of an action.

We will first summarize our approach to solving this problem and then retrace our steps in more detail in the succeeding sections. First, we developed Tapir, a robust action description language that was both more declarative and more succinct than the existing action code. Tapir actions are self-describing: they can be used by planners and executed in simulation. Although Tapir actions were much closer to model components, they were still missing three vital pieces. Firstly, Tapir actions still had to drop down into code to alter the underlying simulation. Secondly, Tapir actions were still monolithic – each action handled only particular circumstances and there were no mechanisms to correctly

sequence and choose related actions as circumstances changed. Lastly, Tapir actions still had no sense of what they were meant to accomplish. In summary, they were syntax without semantics. To address these three issues, we developed a more declarative simulator substrate (Krill), added simple reactive planning to Tapir and created a graphical language to describe desired effects (Action Models).

These language additions allow for the creation of CtF actions that can be readily combined and evaluated. By themselves, however, they do not add the necessary *mental* models into CtF. To do this, we extended our initial morale model and incorporated it into CtF. Having units with morale allows commanders to plan to defeat the enemy by reducing their will to fight rather than just by pure attrition.

Simultaneously with the changes being made to CtFs modeling ability, we also extended it in numerous directions in order to make it a more flexible and general tool for military simulation. These enhancements included:

- Terrain improvements including buildings, bridges and better roads
- Improved sensor models for fog of war
- Line of sight models
- Pluggable overlap detection system in 2 or 3 dimensions
- Network support for multi-user scenarios
- More unit types including civilians and militia
- Better firing models
- Completely new OpenGL based graphics framework including transparency, texture mapping, etc.

Because many of these changes occurred simultaneously with the work on Tapir and Krill, the CtF system is now an amalgam of multiple paradigms; not all of the changes listed fit into the full frameworks used by Tapir and Krill (which will be described below). First, however, we provide an overview of CtF extensions:

Terrain improvements

Capture the Flag has always been a 2.5-dimensional war game (i.e., two-dimensional with varying surface terrain plus some air based units). Originally, the terrain was represented using the same two-dimensional grid that served to increase the speed of overlap detection. Unfortunately, the needs of overlap detection and those of terrain representation are in conflict: the overlap detection grid wanted to be coarse whereas the terrain representation wanted to be fine. Secondly, it was impossible to represent rivers or roads well on a grid. We solved these problems by separating overlap detection from terrain representation; overlap detection is discussed in more detail below. All terrain in CtF is now represented using polygons (i.e., both surfaces such as forest or urban and

features such as rivers or roads). Polygons are not confined to a simple grid and therefore are flexible enough to model interesting and realistic terrain at varying scales.

Improved sensor models for fog of war

CtF originally assumed that information about friendly units was immediately known. It also assumed that all information learned about enemy units was immediately available to all other units. These were reasonable assumptions for the large scale force on force warfare that CtF modeled. As we entered smaller scale combat, we found the need to rewrite our sensor models to better accommodate line of sight and to provide the potential for communication delays between units. We also added additional realism by making the amount of information a unit can obtain dependent on its type, terrain, fatigue, and status (e.g., a unit under fire is much less likely to notice other enemy activity).

Line of sight models

When CtF was used to model large force on force simulations, line of sight was not a significant concern. The level of detail was coarse, blobs represented large groups and terrain features did not alter sighting. This all changed, however, when we began to model small scale operations in urban settings. Here, the ability to hide behind a building or peek around corners becomes central. It is easy to check if unit A can see unit B by "ray-tracing" from the A to B. The difficulty is that ray-tracing takes time and, since we must potentially know about every pair of interactions, we must ray trace on the order of n^2 times every tick. To ameliorate the computation, we added a line-of-sight (LOS) grid to cache the computations (see Figure 1).

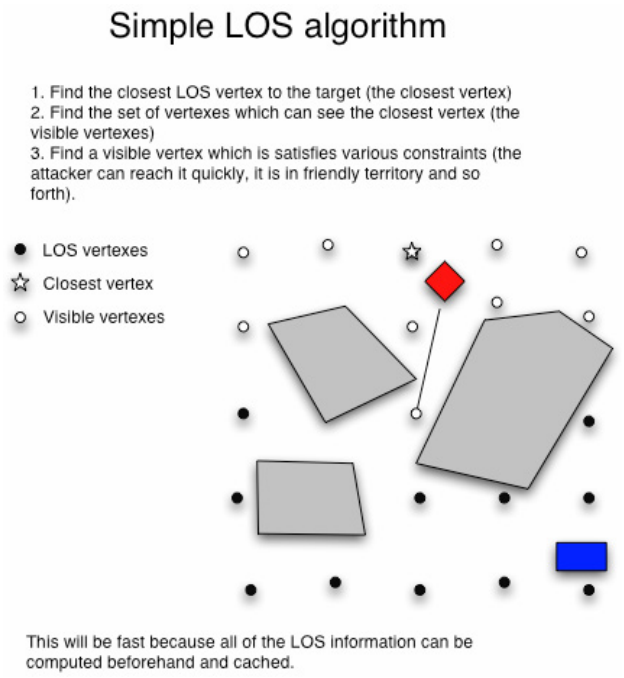

Figure 1: LOS Implementation using ray tracing

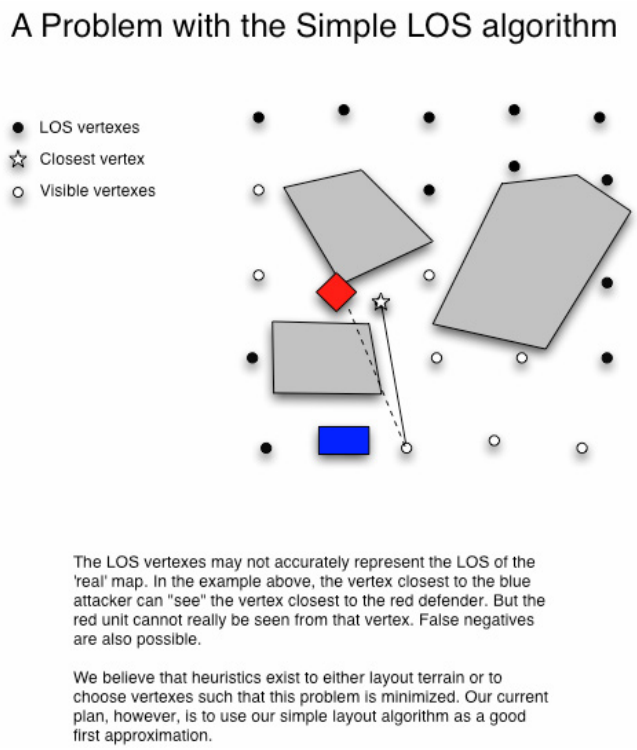

Figure 2: Potential problems with fixed LOS grid

This reduces overall accuracy (since the grid is necessarily coarser than the simulated world) but does allow us to model more complex actions and interactions (see Figure 2).

Pluggable overlap detection system in 2 or 3 dimensions

As mentioned above, the original CtF system used an overlap detection grid to help speed overlap computations. Krill introduced a more abstract overlap detection layer as a new API that was more flexible and allowed for "pluggable" models and experimentation. We have implemented three-dimensional Voroni-based overlap detection and experimented with other two-dimensional schemes based on both grids and spatial coherence.[1]

Network support for multi-user scenarios

Earlier CtF projects included rudimentary networking support. This included the ability to connect CtF to commercial real time strategy games such as BattleZone and Dark Reign and preliminary work on a browser based Java CtF client. Our more recent

[1]Spatial coherence simply means that things do not change very much in small amounts of time so that it is possible to keep sorted lists of positions in roughly linear time. Unfortunately, this technique did not improve upon our existing grid based overlap detection code.

network support includes more scalable peer-to-peer multi-player support. This allows several people to control each side (blue or red) and for observers who can see the action but not control it. Because there is not one single server, the amount of network traffic is greatly reduced and the computation load is shared among all playing machines.

More unit types including civilians and militia

We extended the unit types to include civilians and militia by adding to the class model. These units have different firing models, different fatigue and defeat models, and different planning models (See below for more information on the models we use).

Better firing models

The original CtF system employed three force models: force was applied through close combat (overlap), direct combat (proximity) and in indirect contact (artillery). The first two were based on the physical overlap and proximity of units; the last on an action at distance model where artillery blobs could project force into distance regions. Indirect force used an abstract "spotter" model which led to an initial dispersed and ineffective attack that quickly became more focused as the "range" was determined.

As the scale of our models decreased, it no longer made sense for direct fire units (tanks, infantry and the like) to be able to attack every other close by unit (as they did with the overlap model). Thus we further abstracted our artillery model and used it for small scale direct fire as well.

OpenGL based graphics framework

CtF graphics were based on the Macintosh QuickDraw architecture. We recreated the graphics layer using OpenGL because Apple computer was deemphasizing QuickDraw and emphasizing OpenGL, and because OpenGL is a portable graphics standard. Our current implementation is still tied to the underlying OS X windowing layer but because much of what we do is now OpenGL based, porting to other architectures will be significantly simpler. Furthermore, OpenGL is a more mature graphics framework and this lets us make use of transparency, alpha channel blending and other visual enhancements.

2.0.1 Tapir: A better action language

Tapir is a general purpose, agent control language that extends and enhances the Hierarchical Agent Control (HAC) architecture (see Appendix 2). Tapir incorporates the lessons learned from developing HAC and makes it easier and faster to create reusable and understandable actions. Tapir can be used for simple scripting and can easily handle reactivity and more complex planning. Tapir has controlled units in a battalion level wargame, simulated robots, and simulated rodents. The language is built around constructs that define agents, sensors, actions, and messages. It has mechanisms for handling multiple agents, a flexible resource model, and multiple means for structuring concurrent actions.

The following example shows how the move-to-point action is described in Tapir. Appendix 3 shows the same action as implemented before Tapir. Though still complex, the Tapir action is significantly simpler and takes 4 times *less* code.

```
(defaction move-to-point ()
 (:documentation "Moves a blob so that it is within the target-geom.")
 (:parameters ((target-geom nil :export :type (or null geom))
               (terminal-velocity 0 :export)
               (terminal-facing nil :export)))
 (:estimated-cost (:time (move-to-point-estimation simulation action))
                  (:mass 0))
 (:satisfies ((reconnaissance-goal :target-geom target-geom)))
 (:sensor (path-planning-sensor
           :team (team blob)
           :messages-to-generate '(path-planner-new-path-message)))
 (:resources ((blob :count :all :part-type blob-motion-resources
                    :type basic-military-blob)))
 (:locals ((path-edges nil)
           (est-time-to-complete nil)))
 (:on-message (failure :send-message-to-parent))
 (:on-message (completion
               (:debug "MTP: ~A reached ~A" (name blob) target-geom)
               (:generate success)))
 (:on-message (path-planner-new-path-message
               (:debug "MTP: ~A, new path message." (name action))
               (setf path-edges (path the-message)
                     est-time-to-complete (path-cost the-message))
               (stop-children)
               :restart))
 (:do
   (:case (whole-resource-count blob)
     (1
      ;; single blob
      (:debug "MTP: Sending ~a (~,2F, ~,2F) to ~a via ~a (ETA: ~,0f)"
              (name blob) (x (location blob)) (y (location blob))
              target-geom path-edges est-time-to-complete)
      (move-with-waypoints
       :blob blob
       :waypoint-list
       (mapcar (lambda (edge)
                 (make-destination-geom (location (vertex-to edge))))
               path-edges)
       :target-geom target-geom)
      (:when terminal-facing
        (turn :blob blob :target-facing terminal-facing)))
     (:otherwise
      ;; multiple blobs
      (formation-move-to-point
       :axis axis
       :target target-geom))))
 :export)
```

Without describing the Tapir language in detail (see Appendix 4 for a manual), we can see that the action takes a *blob* as a resource and has a *target-geom* parameter.[2] In words,

[2]In the Abstract Force Simulator (AFS) and Tapir, geoms are abstract two-dimensional regions, and resources are something that an action can use to get things done. Blobs are the CtF units. They can be individuals or group blobs. The *whole-resource-count* accessor is used to tell how many individual units compose the blob.

the move-to-point action should get the blob to move to the target-geom. The actions **:do** clause describes how the action should accomplish this. The move-to-point action will either call on formation-move-to-point if its blob resource is a group blob or it will call on move-with-waypoints if it has an individual blob. The **:on-message** clauses send success and failure messages up to the move-to-point's parent action (if any) and deal with changes in the current best path. These changes are monitored by the path-planning-sensor.

Move-to-point shows how more complex actions can be composed from simpler ones. By adding sensors and message handlers, Tapir actions can arbitrarily extend or override the behavior of their parts.

Tapir provided a huge leap forward in action writing productivity. It became much easier to build and debug complex actions because Tapir took care of so many of the details. As the following actions show, however, all was not perfect. The primitive-move-to-point action is the lowest level movement action. It is in charge of actually setting a blob's acceleration effectors so that it will turn and move.

```
(defaction primitive-move-to-point (move-to-point-mixin)
  (:resources ((blob :count 1 :part-type blob-motion-resources
                     :type basic-krill-blob)))
  (:parameters ((target-geom nil :export :type (or null geom))
                (terminal-velocity 0 :export)
                (terminal-facing nil :export)))
  (:estimated-cost (:time (calculate-time-length
                           simulation blob
                           (location blob)
                           (location target-geom))))
  (:wake-if (and blob target-geom
                 (at-destination-p the-action target-geom))
            (:generate success))
  (:do
    (:in-parallel
       (:in-sequence
        (:debug "PMtP: turning to ~,2F" (geom-angle blob target-geom))
        (turn :target-facing (geom-angle blob target-geom)))
       (:repeat
        (:code
         ;;; this sets an effector and therefore stops the action
         ;;; until the effector is 'cleared' by advance-physical-processes
         (speed-controller the-simulation the-action))))
    (:when terminal-facing
      (:debug "PMtP: turning to terminal facing of ~,2F" terminal-facing)
      (turn :target-facing terminal-facing)))
  (:on-resource-unbound
   (set-acceleration-effector the-action blob (zero-vector 2)))
  (:print "PMTP: ~A to ~A" (name blob) target-geom)
  (:short-name "PMTP ~A" (name blob))
  (:debug-superclasses move-to-point-actions)
  :export)
```

The most important part of primitive-move-to-point, however, is hidden in the call to the speed-controller function. This function takes care of all the interactions with the underlying simulator and actually sets the blob's effectors. Thus the action does not

provide any declarative information about what it does that can be used by other programs and agents.

Move-to-point is also problematic because it does not take the context of the movement into account. For example, if the blob is moving in service of a screening action and it comes across a small enemy force, then it should engage and destroy it. If the enemy force is large, however, the blob should avoid it. If an enemy attempts to engage the blob, it should fight back or flee (depending on its mission and the enemy's strength). None of these conditions are included in move-to-point – indeed; they should not be because they have nothing to do with general movement. It is possible to write Tapir actions that handle all of these contingencies and wrap calls to move-to-point within them. The combinatorics of this quickly becomes unwieldy, however, and we knew that a better solution was required.

Lastly, these actions still do not describe what they are meant to do and how one can tell if their goal has been achieved or how well it has been achieved. We next describe three tools we implemented in order to fix these three problems.

2.0.2 No More Code: Declarative Executive Simulation

Tapir actions had to drop down to code because the Abstract Force Simulator (AFS), the simulation substrate upon which CtF is built, was not declarative. Like the original CtF actions, AFS could be understood by programmers but did not expose its working in a fashion that other computer programs could use. For example, although friction in AFS controls how quickly units move. The implementation of friction was opaque; there was no mechanism to express that friction and movement rates were related and the relation itself was spread among multiple classes and objects.

Krill is a new simulation substrate that explicitly declares agent sensors, properties and effectors. In Krill, we can express the following facts:

- Change in location arises from velocity
- Velocity arises from acceleration
- Acceleration arises from agent effectors and friction
- Friction arises from terrain, overlap, reduced morale and damage

These facts allow a planner to deduce that it can slow an agent by increasing its friction and that it can increase friction by overlapping the unit, altering its terrain, damaging it or reducing its morale.

Krill also takes on some other AFS responsibilities such as overlap detection (in 2 or 3 dimensions), generic display routines and some parts of scenario definition. AFS now rests on top of Krill, adding the particular physics used in CtF.

2.0.3 Flexible Actions: Goals and the Tapir Reactive Planner

The second problem we addressed was the difficulty of making actions sensitive to context. As discussed in the move-to-point example above, blobs need to react intelligently to changes in terrain and other units (both friends and enemies). Writing individual Tapir actions that correctly handle all contingencies is extremely difficult because there are so many possibilities and new ones arise all the time. Instead, we added explicit goals to Tapir and developed a simple reactive planner to let actions flexibly react as circumstances dictate.

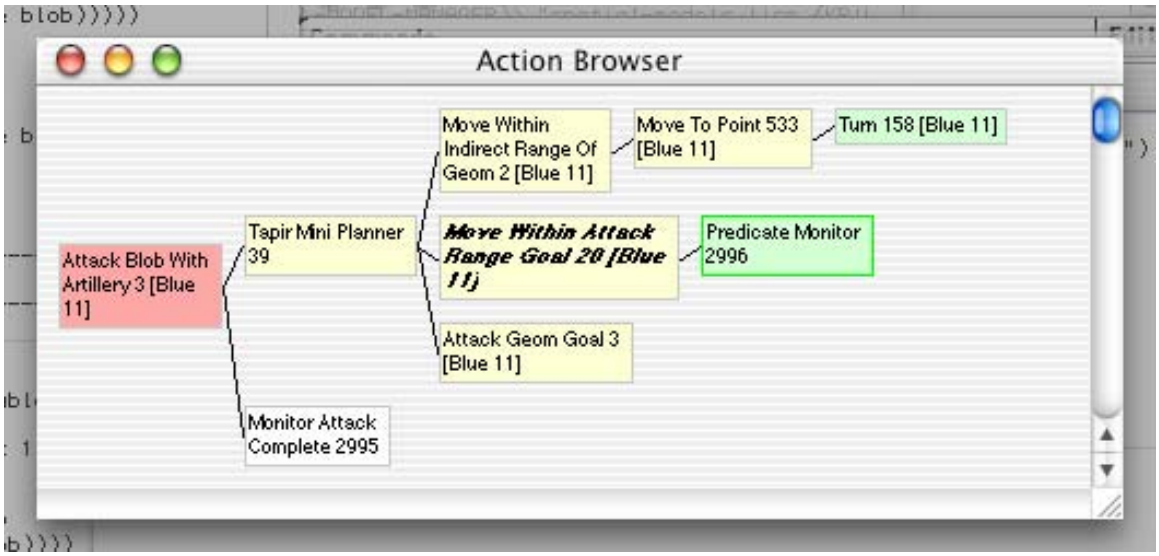

Figure 3: Action Brower showing an Attack Blob with Artillery action

The action browser in Figure 3 displays one moment in the life of an Attack-blob-with-artillery (ABwA) action. Since the actions: **do** clause posts goals, a mini-planner has been created as a child. The mini-planner has AbwA as its parent and two goals as its children: Attack-Geom and move-within-attack-range-goal (MWARG). The Attack-Geom goal's job is to determine the location of the enemy blob and to use the artillery blob's firing effectors to direct fire onto it. The MWARG's job is to ensure that the artillery blob is within range by using its movement effectors. Note that each goal can independently use the different effectors of the artillery blob without conflicts arising.

In the diagram above, the artillery blob is out of range and it is therefore impossible for the attack geom goal to activate. Instead, the move-within-attack-range-goal is active (indicated by the bold type). This goal determines that the best plan to satisfy it, given the current circumstances and the types of its resources and targets, is to move-within-indirect-range-of-geom (MWIRG), the spatial area of its target. MWIRG starts a move-to-point action which starts a turn action. All of this activity occurs during a single tick of simulated time. As the situation unfolds, the mini-planner dynamically starts, pauses, restarts and ends actions associated with its goals. Furthermore, the attack-blob-with-artillery action continues to monitor its own success conditions and may add and remove goals as necessary. Once the artillery blob comes within range, the Attack Geom goal

will also become active and the artillery blob will begin firing. The mix of firing and moving will happen automatically and does not need to be scripting directly into the actions because the mini-planner takes care of the details. This is *significantly* simpler for the action writer.

```
(defaction attack-blob-with-artillery ()
  (:documentation "Attack a military blob with an indirect-force-capable
blob.")
  (:resources ((blob :predicate (indirect-force-capable? blob) :count 1)))
  (:parameters ((target-blob :required :type 'military-blob
                             :predicate (not (or (flag-p target-blob)
                                                 (airborne target-blob))))
                (mission :value :destroy)))
  (:do
    (:in-parallel
     (:maintain (move-within-attack-range-goal :target target-blob))
     (:maintain (attack-geom-goal :target target-blob :mission mission))))
  (:wake-if (:nickname attack-complete)
            (attack-termination-p)
            (:generate completion)))
```

Two keys points about this snapshot are the dynamic / real-time nature of Tapir planning and the declarative nature of Tapir action descriptions. The ABwA description specifies only what goals to maintain. The programmer does not need to consider sequencing or which actions are more likely to achieve the goals. Nor do they need to worry about new means of goal achievement – if, for example, a teleporting artillery unit was available, MWARG could be achieved in a miraculous fashion but the ABwA action would not need to be re-written to take advantage of the new capability.

2.1 Capture the Flag in Action: operation Northampton

The improvement mentioned previously to the simulation engine, to Tapir and to Krill can be seen in the Figures 4 and 5.

The scenario pitted blue infantry and attack helicopters against a much larger but disorganized force of red militia in a Somalia like situation. Tapir reactive plans controlled the red militia causing it to swarm at blue units and especially at any downed helicopters. Damaged red units would automatically retreat and attempt to regroup. Blue units could be controlled by the human player, by the planner or in mixed initiative fashion with the human controlling the infantry and the computer taking control of the helicopters. The close up in Figure 5, shows that situations could rapidly get out of hand.

This scenario provided an excellent test bed for many of our new technologies including Action Models and Defeat Mechanisms which we turn to next.

Figure 4: Red militia against Blue infantry and helicopters in a Somalia like scenario on the outskirts of Northampton, MA

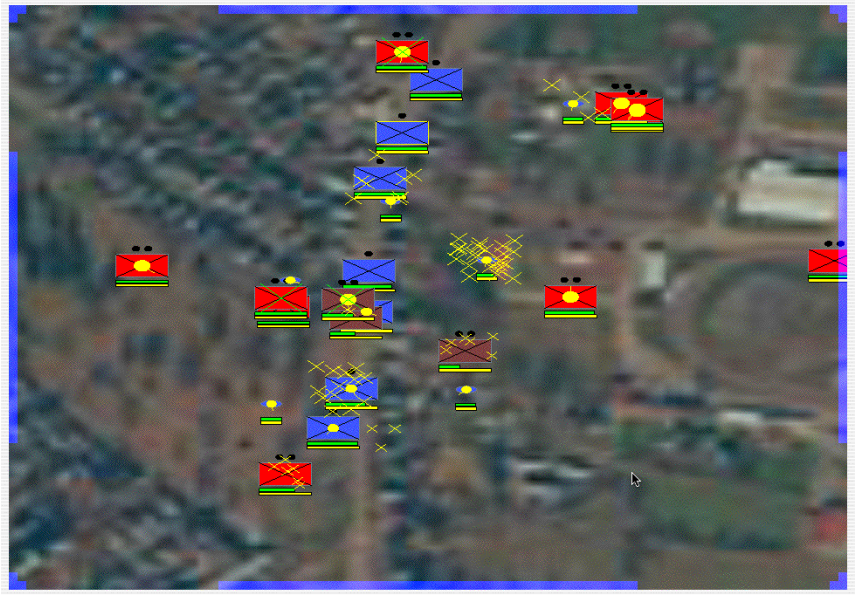

Figure 5: Close up of Northampton scenario unfolding

2.2 Actions and Effects: Action Models

Building military simulations requires bridging the gap between the knowledge of commanders and computer representations of that knowledge. A significant part of this knowledge concerns military tasks, their interactions, and an understanding of how to

grade their achievement. *Action Models* describe the complex spatial and temporal dynamics of goal directed tasks with a graphical notation. Commanders can understand the notation and Knowledge Engineers can convert it into declarative or procedural forms. The conversion makes possible automated After Action Reviews of plans written in terms of these tasks [10]. We now describe Action Models and grading scales.

Evaluating an action requires measuring and grading the performance of its sub-tasks. An Action Model describes the sequence of an action's expected events and goals using a simple graphical notation. Action Models idealize both the temporal and spatial aspects of an action. In doing so, they provide a bridge between the language of Subject Matter Experts (SMEs) and Knowledge Engineers.

Action Models consist of an abstract *graphical depiction* of an action's spatial interactions, *anchor points* (indicating transitions between phases of the action) and *attribute timelines*, specifying measurable attributes. For example, Figure 6 depicts the Action Model for the **defeat** task from Army Field Manual FM3-90 [11].

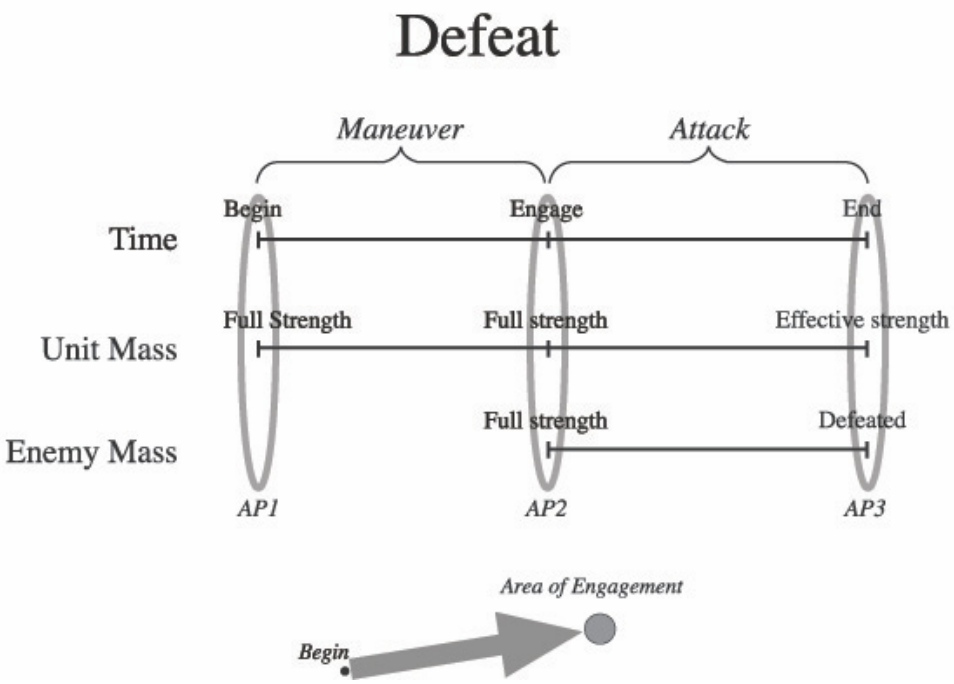

Figure 6: The Action Model for Defeat

The Action Model for **defeat** contains three anchor points, dividing the **defeat** action into two phases: a maneuver phase and an attack phase. The anchor points define the beginning of the **defeat** action (AP_1), the transition from maneuver to attack when the enemy unit is encountered (AP_2), and the completion of the action (AP_3) when the enemy is defeated, retreats, or the attacking unit is destroyed. Measurable attributes are time, the friendly unit(s) firepower, and the enemy's firepower. Time and friendly firepower can be measured at all three anchor points, but enemy firepower is only measured at the

beginning of engagement and at the completion of the **defeat** action (AP_2 and AP_3, respectively).

Although Action Models are general and can be used to model any activity with well-defined measurement criteria, we have concentrated our work in the military domain. In particular, our Action Model examples are drawn from Army Field Manual FM3-90 and have been used in Course of Action (COA) construction and evaluation.

Evaluating an action requires grading the performance of its sub-tasks. An Action Model describes the sequence of these tasks and the desired values of key attributes at their beginning and end. It also provides a spatial model of the task which represents abstractly how the tasks and the attributes are related.

Attribute measurements are graded with the five-point scale (Figure 7) used in After Action Reviews.

Figure 7: Five point grading scale

The Action Model for **defeat** (Figure 6) shows the desired attribute values at each anchor point. For example, the unit's firepower should be at least *effective* and the enemy's firepower should be *defeated* at the end of the action.

Different attributes require different kinds of grading scales. Two useful scales in the military domain are mass and timing. Mass is an abstraction of all the characteristics of a force that make it effective. It includes force size, weapon type, cohesion, morale, training, experience and so on. Timing is the coordination of tasks relative to a single action and to a plan as a whole.

Mass Grading Scales

Mass scales measure the effectiveness of military units relative to their initial strength. The effectiveness of a unit depends both on its percentage of remaining mass and on how well it can cope with the loss of that mass. For example, elite units will continue to function effectively with much higher casualty levels than green troops. To account for this, we use separate scales for unmotivated, veteran and elite troops. Figure 8 depicts the scale for veteran troops.

Each scale maps the percentage of remaining mass to a military effectiveness level: full strength, neutralized, defeated and so on.

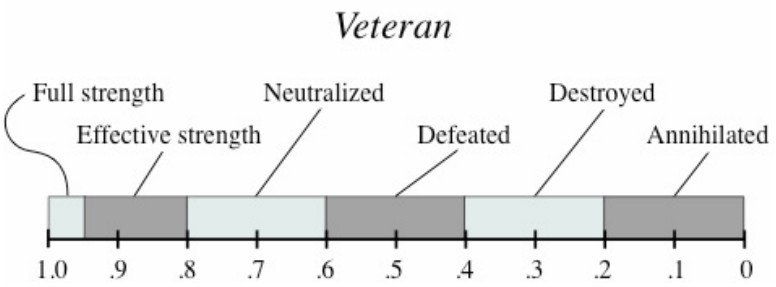

Figure 8: Mass scale for veteran troops

The labels on the mass attribute timelines (Figure 6) specify desired effectiveness. A unit at this level would receive a grade of **OK** (Figure 7) -- if the unit is at another level, its grade would depend on whether it is a friendly or enemy force. For example, if a friendly blue unit is trying to **defeat** an unfriendly red one; then the final goal (AP_3 in Figure 6) is to have the blue unit at **effective strength** and the red unit at **defeated**. For veterans, the 5-point scale is calibrated so that a final mass between 80 and 95% of initial mass will receive a score of **OK**. Mass ranges around this base take on other scores. For example, because we are measuring friendly mass, levels higher than 95% will receive a score of **+**, but if mass is in the **neutralized** range (60 to 80%), a score of **-** will be assigned. Even lower masses will receive a **--**. On the other hand, if we are measuring desired negative effects on an enemy unit, and the goal is to reduce its mass to **defeated**, then an **OK** would be assigned if the final enemy unit mass is between 40 and 60%, a **-** if between 60 and 80% (neutralized), a **--** if between 80 and 95% (effective strength), **+** if between 20 and 40% (destroyed), and **++** if between 0 and 20%.

A final note: mass could be measured two ways, unit-based or terrain-based. Unit-based means, that the mass of one or more units is being measured. In the case of a target *group* of units, the sum of masses of each unit at the beginning determines the initial **full-strength** of the group, and the percentages are determined by the ratio of total remaining mass to that initial full-strength. Terrain-based means, that all of the enemy mass within a spatial region is summed and reported. Similar to the way groups of multiple units are handled in unit-based measurement, the mass of the units remaining in the region at the time of measurement is summed and compared to the sum of their initial masses.

Time Grading Scales

Inter- and intra-action coordination is assessed using time scales. Time may be measured on an absolute scale or relative to the timing of sub-tasks. As mentioned, Action Models have internal structure based on transitions between sub-tasks. For example, the **defeat** action has a beginning, an engage-event, and an end -- these are the events which a commander can use to coordinate with other events or actions in a COA. A *time grading scale* is a way of asserting and assessing coordination goals.

The scale provides an envelope of time intervals around some specific time *t*. Grades are assigned to each interval according to the timing relationship the envelope represents. With these grades, the envelope describes the commander's desire for how close to *t* some

event *should* occur, keeping in mind that there are many variables that may affect whether timing goals specified in a COA can be met in actual practice. Time *t* is a general temporal marker: it may refer to absolute time (e.g., related to specific hour times), to coordination times defined in a COA specification (e.g., by *H*-hour), or to other events (e.g., the completion of action-***b***, the arrival of unit-2, or the crossing of phase-line *Lima*, etc.).

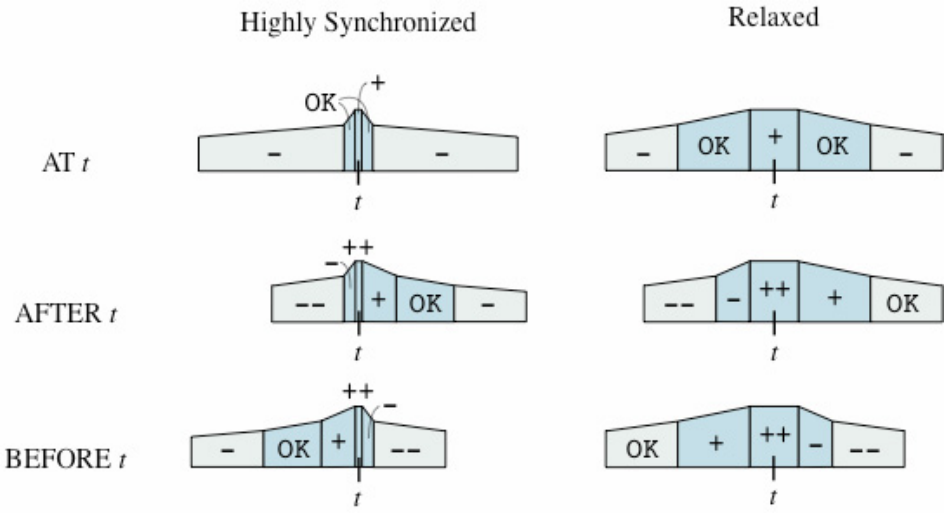

Figure 9: Various Time scales

Figure 9 depicts grading scales for three types of timing relationships between *t* and an event, each with two settings. A seventh option is to assert that timing does not matter for the event. The scales in Figure 9 indicate that the event should happen at *t*. The score received depends on the tightness of the scale, how distant the event is from *t*, and whether the event occurs before or after *t*. If it occurs at *t*, it will be scored as **OK**.

Abstract Spatial Representations

Action Models also include a depiction of the spatial relationships which the action ought to maintain or which can be used to understand the relationship between sub-tasks. The **defeat** action has a very simple model showing that the friendly unit must first travel to the enemy and then engage it. Other models have more complex depictions.

2.2.1 Action Model Examples

The next few sub-sections provide examples of Action Models. Each example begins with a short italicized paragraph describing the intent of the action. These are paraphrased from Appendix B of Field Manual FM3-90. We have also codified Fix, Block, Attrit, Destroy and several other FM3-90 actions.

Screen

> *A unit performing a **screen** observes, identifies, and reports enemy actions. Generally, a screening force engages and destroys enemy reconnaissance elements within its capabilities -- augmented by indirect fires -- but otherwise fights only in self-defense. The **screen** has the minimum combat power necessary to provide the desired early warning, which allows the commander to retain the bulk of his combat power for commitment at the decisive place and time. A **screen** provides the least amount of protection of any security mission; it does not have the combat power to develop the situation. The **screen** should allow no enemy ground element to pass through the screen undetected and unreported.*

As depicted in Figure 10, a screening force is associated with some larger force some distance behind it, so a **screen** action must coordinate with the larger force.

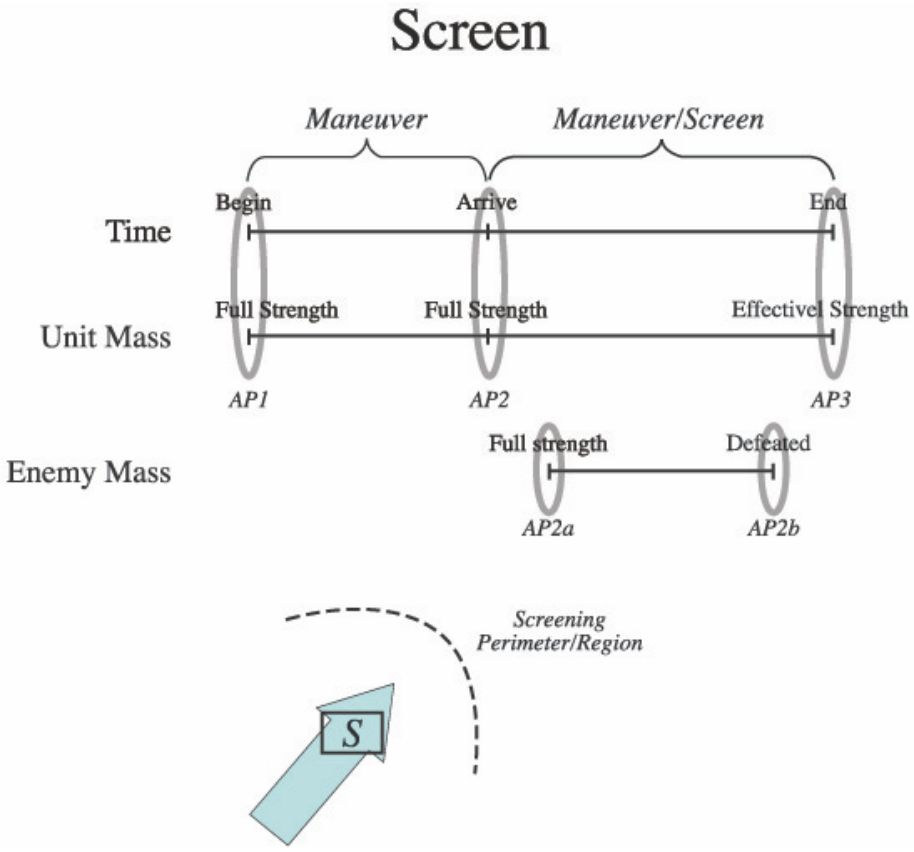

Figure 10: The Action Model for Screen

The screening force is small, and its primary mission is intelligence gathering and counter-intelligence operations. It will only engage small forces being used for intelligence purposes by an opponent -- for our purposes, we can assume that it will only attack small forces for which it has a better than 1-1 combat power ratio. The large arrow in the figure depicts the direction *from* the force that the screening unit is in service of.

Throughout the **screen**, the screening unit may move within the screening perimeter as necessary in order too maintain contact with any enemy units, while also keeping a safe distance to avoid engaging a larger enemy force.

The two main attributes measured are the time for which the screen is to be in place (absolute or relative to other events (e.g., completion of preparation of the force the screen is in service of), and the mass of the screening unit). The screening action has two phases, a travel phase which ends when the unit arrives at the screening location (AP_2), and maneuvering to patrol the area. The screening unit should still be at effective strength by the end of the **screen**. The enemy mass attribute scores how effectively the screening unit managed counter-intelligence operations, if any.

Breach (Penetrate & Exploit)

> *Use all available means to break through or secure a passage through an enemy defense, obstacle, minefield or fortification.*

A **breach** (Figure 11) is an Action Model that involves explicit coordination between two units: a *penetrating* force and an *exploiting* force.

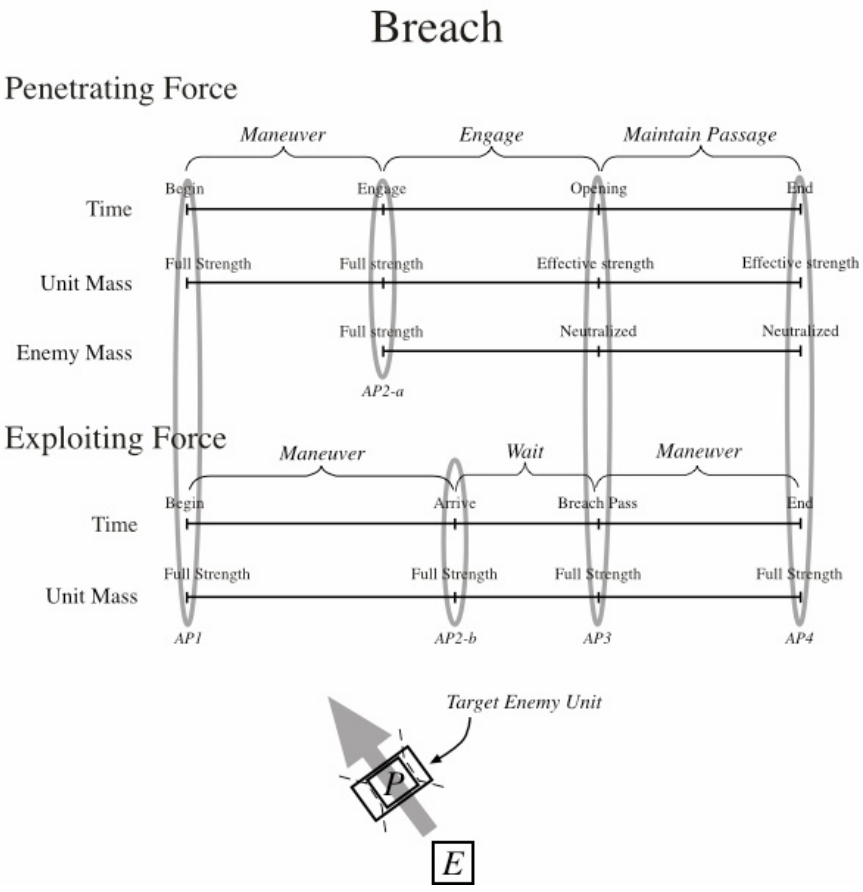

Figure 11: The Action Model for Breech

The penetrating force is tasked with engaging an enemy force in order to create an opening through which the exploiting force may safely pass. The diagram at the bottom sketches what this looks like. The penetrating force (indicated by *P*) overlaps the target enemy unit(s) and creates an opening. Meanwhile, the exploiting force (*E*) maneuvers into position and waits. When the enemy is breached, it moves through the opening. The direction of the movement, indicated by the arrow, is assumed to be in the direction opposite where the penetrating and exploiting force began their action.

A breach action has three phases for each unit (for four anchor points each). The penetrating unit maneuvers to the enemy, engages it to create an opening, and continues to engage to maintain the opening. The exploiting unit maneuvers into a waiting position, waits and then maneuvers through the opening once it is created.

All of the anchor points for either friendly force are synchronized except for $AP_{2\text{-}a}$ and $AP_{2\text{-}b}$ which may happen at different times (in the Figure AP_2 is reached after $AP_{2\text{-}a}$, but this is not necessary); the only constraint is that the exploiting force achieve the best position for exploitation before the penetrating force creates the opening. While the opening is present, it is assumed the target enemy will not be able to affect the exploiting unit, although it may still affect the penetrating force. The goal of the breach is the exploitation of the opening by the exploiting force, not the destruction of the enemy, so the goal is to just neutralize the enemy. The exploiting force should also be at full strength by the end of the breach action.

Action Model Summary

In summary, Action Models are novel formulations of activity. They describe actions with a spatial model, a set of anchor points (dividing sub-tasks) and a set of measurable attribute timelines (describing goals). They help to bridge the gap between Knowledge Engineers and Subject Matter Experts.

3.0 Defeat Mechanisms

In addition to the changes made in CtF's modeling languages, we also investigated several different Defeat Mechanisms and incorporated them into CtF. Defeat Mechanisms are ways of defeating an enemy other than by strict attrition. They function primarily at the mental level. For example, we can defeat an adversary by reducing their morale or by alienating them from the population supporting them.

3.1 Heeringa and Cohen Model

The first model we explored was due to Heeringa and Cohen and is inspired by Gen. (ret.) Keith Holcomb and other literature. It is composed of two components: a Fatigue Model and a Defeat Model. The Fatigue Model combines factors that affect emotional and physical fatigue to produce an overall measure called *effective fatigue*. The Defeat Model combines effective fatigue, along with an agent's state, to produce probabilities of surrender.

The Fatigue Model. We do not simulate psychological or physiological processes in individual warriors, but instead we model the collective fatigue of a unit (e.g., a battalion) as a weighted sum of factors that influence fatigue. Fatigue does not refer exclusively to physiological fatigue, but rather to physical, emotional, and personal components. Physical fatigue can be thought of as a depletion of energy or mass, while emotional fatigue summarizes the effects of fear, courage, aggression, and morale.

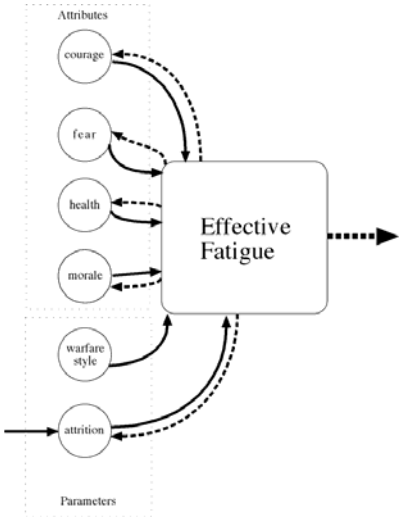

Figure 12: Feedback model for effective fatigue

A sketch of our model of effective fatigue is shown in Figure 12. Note that this is a feedback model in which attributes such as morale both affect and are affected by effective fatigue. The model also implements a mental-physical feedback loop. Notice that attrition has two inputs, one from the external world and one from effective fatigue; and one output, pointing to effective fatigue. This means attrition levels increase faster with effective fatigue, and contribute to effective fatigue. Such a system is inherently unstable. Under some conditions, physical effects (attrition) and mental effects (effective fatigue) can increase faster and faster in a vicious cycle. In practice, parameters of our model damp this acceleration, but it is there, albeit attenuated, and is a good example of the nonlinearities that can occur in warfare.

The Defeat Model. The output of the Fatigue Model is one of the inputs to the Defeat Model. The Defeat Model contains a base probability of surrender, a set of states, rules for specifying when state transitions are made, and functions that specify how the current probability of surrender is computed based on the time spent in the current state. Every Defeat Model has an initial base probability of surrender. This is purely a function of effective fatigue. We chose an exponential function, meaning that the probability of surrender increases exponentially fast with level of effective fatigue. In addition, modelers may define other states. These modify the initial probability to produce an agent's final probability of surrender.

3.2 Searching for Simplicity: The Bucket Model

We evaluated the Heeringa and Cohen Model above and found that it could produce interesting behavior with clear non-linear dynamics. However, we felt that the model was too complex and that the large number of parameters made it unwieldy. We then turned to a simpler model that viewed fatigue as a leaky bucket: an agent's fatigue rises when it is damaged and especially when its enemies damage it more than it damages them (see Figure 13).

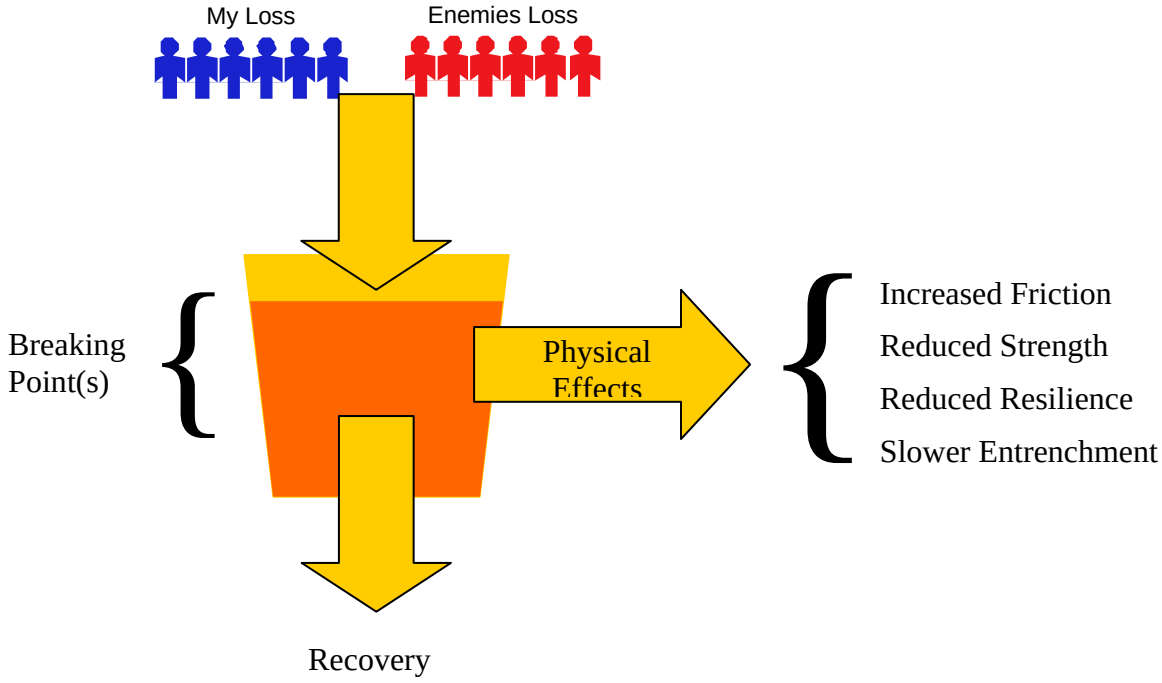

Figure 13: Defeat as a leaky bucket

The fatigue also has a natural constant recovery rate. The fatigue has a linear effect on the effectiveness of the agent where effectiveness is modeled by scaling the agent's key properties away from their nominal values. Though it is not explicit, this model is non-linear because as the fatigue rises, the effectiveness falls and as the effectiveness falls; the agent is liable to take more damage (and dole out less) which will cause the fatigue to rise more quickly. More explicitly, we model the fatigue at time *t* as a function of the fatigue at the previous time plus some inflow *f* and minus an outflow *g*:

$$F_t = F_{t-1} + f(F_t) - g(F_t)$$

Each effect at time *t*, $E_{t,i}$, is modeled as some function, h_i, of the current fatigue level.

$$E_{t,i} = h_i(F_t)$$

We choose simple functions for *f*, *g*, and *h*:

$$f(t) = \Delta M_s + \Delta M_s / (\Delta M_s + \Delta M_e)$$

$$g(t) = R$$

$$h_i = P_i(1 + k_i F_t / B)$$

Where ΔM_s is the change in a unit's own mass, ΔM_e is the total change in the mass of any enemies it is attacking, R is a constant recovery factor, B is the bucket size, k is a maximum effect size (e.g., 30%), and P_i is a scaling factor. These equations mean that all increases in fatigue are based entirely on attrition and that they are linear with the proportion of total fatigue to total bucket size.

We evaluated the Bucket Model by implementing it in CtF and then seeing how blob behaviors were altered. When the abilities of the blobs were affected by their morale, the following changes were observed:

- Changes in the overall length of battles
    - These could be longer because, by analogy, weary boxers do less damage…
    - They could also be shorter because the blobs sometimes gave up
- The total attrition suffered (by both sides) was reduced
- A small force can defeat a much larger one if the larger one is demoralized
- Coordinated attacks matter even more. Figure 14 and 15 show how an uncoordinated attack fails to destroy the enemy because there is time for him to recover between the attacks. A coordinated attack, on the other hand, does not give the enemy time to recover and so destroys him. These Figures were created using a CtF scenario where two smaller blobs attacked a much larger one. In the first case the attacks occur directly after one another and cause the larger blob to surrender (its fatigue goes about 100%). In Figure 15, the second attack is delayed and the larger blob has time to recover.

The Bucket Model adds morale to CtF using few parameters and it produces behaviors essential for EBO.

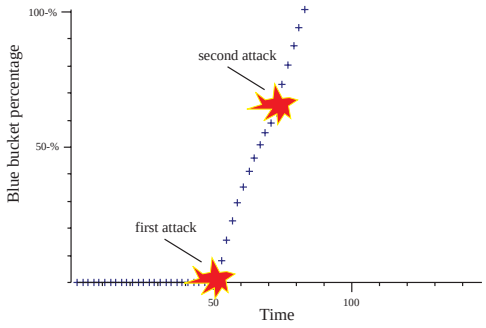

Figure 14: A coordinated attack destroys the enemy

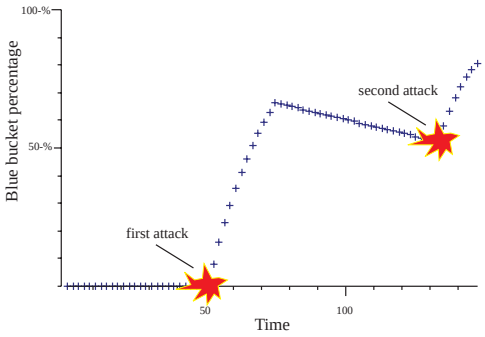

Figure 15: An uncoordinated attack fails to destroy the enemy because he has time to recover

3.3 Modeling Defeat as Disorganization

We also explored a different notion of defeat based on disorganization and entropy. All organizations must adapt to their changing situation and environment. In a hierarchical organization such as the military, this includes reacting to the immediate situation and communicating changes both up and down and across the chain of command. If things are going according to plan, then the communication needs will be small: "everything is going as planned". On the other hand, things can go wrong in all sorts of ways causing re-planning at each level of the system and the need to communicate both changes in the environment and changes in internal structure and goals. This notion is behind the idea of maintaining the initiative and why surprise can be both so effective and so demoralizing.

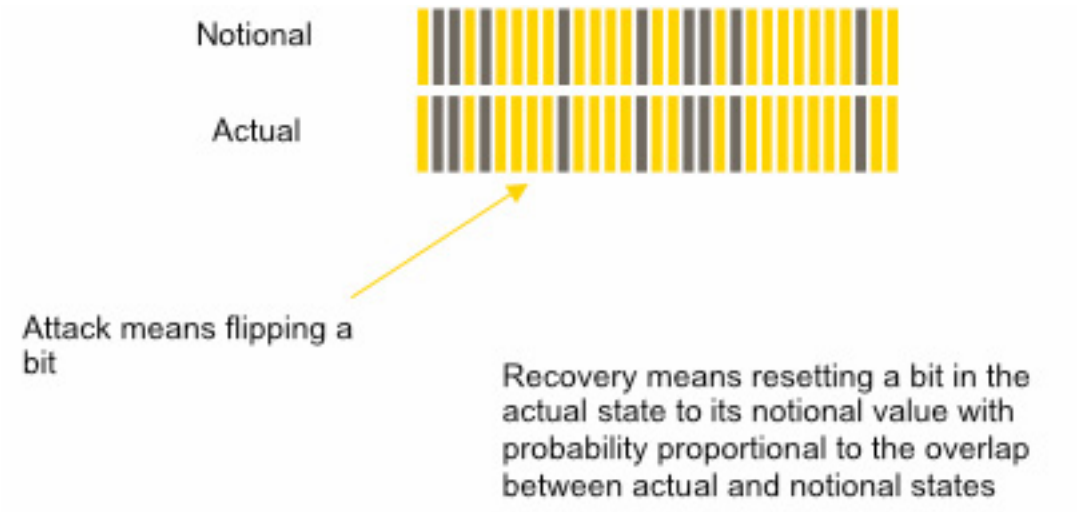

Figure 16: Modeling defeat information theoretically

We explored these ideas in a very abstract simulation based on bit vectors. The goal of the system is to maintain coherence between bit vectors at all levels. The goal of an adversary is to disrupt these bit vectors causing disorganization and, eventually, defeat.

Each level of the system represents their own *notional* view of how things are, which matches more or less with the actual situation. An attack means flipping a bit in the vector that represents the state of the world and recovery means matching the notional state back to the actual state. Recovery is not automatic: when a system attempts to recover, it does so only probabilistically. The probability of recovery is proportional to the overlap between the actual and notional states: recovery is more likely when the system's view of reality is correct.

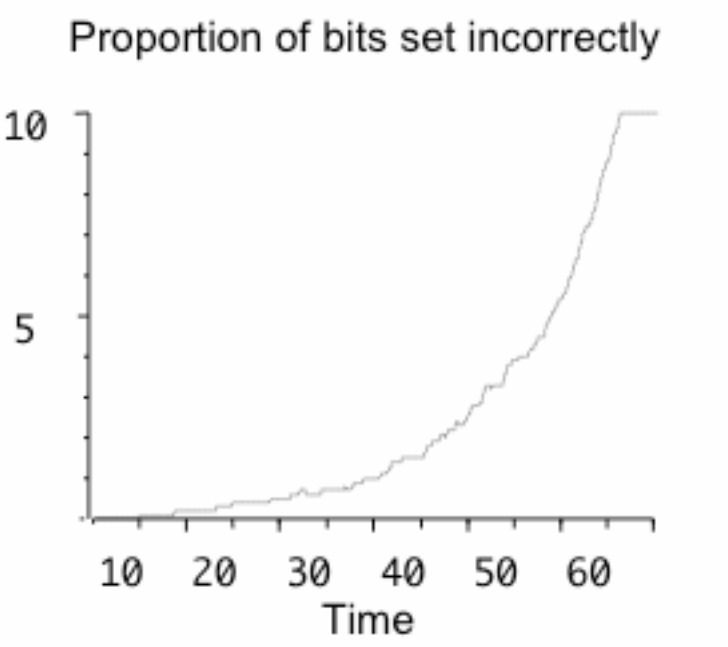

Figure 17: Single level system under attack. As more bits flip out of alignment, it becomes harder and harder to recover.

Figure 17 shows the proportion of bits set incorrectly over time in the single level system of Figure 16 as an enemy randomly flips bits in the actual state. The initial slow loss of coordination accelerates as the notional situation differentiates further and further from the actual situation.

Things become more interesting if we place systems within an organization of their own.

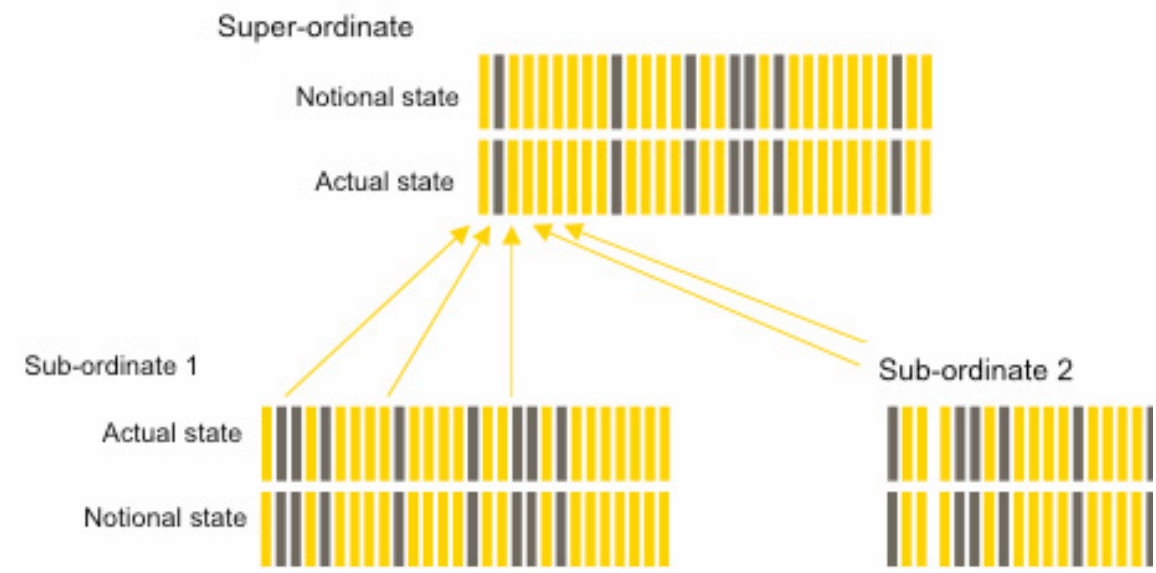

Figure 18: A small system of inter-related systems

Figure 18 shows a small system made up of one super-ordinate individual with two subordinates. Some of the bits in the super-ordinates actual vector are functions of bits in its subordinates. The super-ordinate and its subordinates also need to communicate changes to one another. In our simulation, attacks are made only on the subordinate levels. The super-ordinate is not attacked directly but still must cope with the changes in its subordinates. Interestingly, enemies now have two strategies: they can attack randomly or (assuming that it is known) they can attack the bits of the subordinates that directly affect the super-ordinate. Figure 19 demonstrates the difference these two strategies can have.

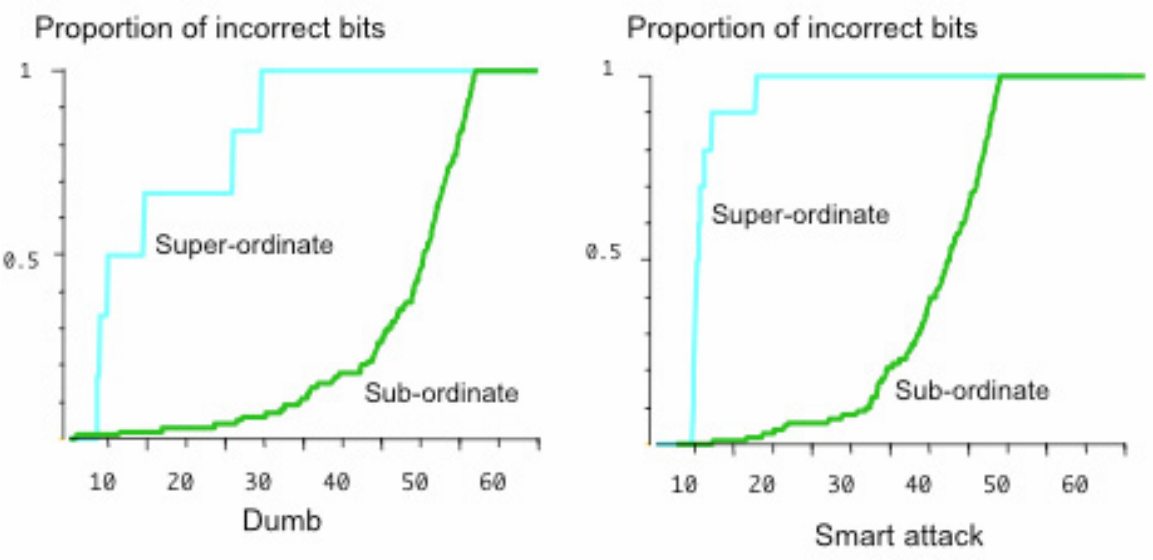

Figure 19: The results of two attack strategies on a multi-level system

Our work in information theoretic models of defeat is only at its early stages but it none-the-less shows how simple models can provide and represent complex non-linear dynamical behaviors. We believe that models such as these are deserving of further study.

4.0 Summary

EBO requires reasoning about the effects of action. To do this, we need models of effects. These must distinguish the physical from the behavioral and physical structures from mental ones. The original Capture the Flag system modeled only physical force on force conflict. The changes we made to CtF under this program greatly enhance the kinds of conflict that can be modeled: from large-scale military engagement on open terrain to squad level scenarios in an urban environment. Action Models and Defeat Mechanisms provide CtF with a much broader range of measurable effects. Krill and Tapir make it possible to describe plans and actions declaratively, while also supporting Monte Carlo simulation for the measurement of uncertain, probabilistic and dynamical events. Together these changes bring effects based models to CtF.

References

[1] ATKIN, M. S., KING, G., WESTBROOK, D., HEERINGA, B., HANNON, A., AND COHEN, P. R. Hierarchical Agent Control: A framework for defining agent behavior. In *Proceedings of the Fifth International Conference on Autonomous Agents* (2001), ACM Press, pp. 425–432.

[2] KING, G. LIFT - LIsp Framework for Testing. In *Proceedings of the International Lisp Conference, 2003* (New York, NY, 2003).

[3] KING, G., AND HANNON, A. The Plan Acquisition Tool: A tutorial. Tech. rep. 125, University of Massachusetts Computer Science, 2003.

[4] KING, G., HEERINGA, B., WESTBROOK, D., CATALANO, J., AND COHEN, P. Models of defeat. In *Proceedings of the 2002 Winter Simulation Conference* (2002).

[5] KING, G., MORRISON, C., AND COHEN, P. Action models. In *Proceedings of the 2003 Winter Simulation Conference* (2003).

[6] KING, G. W., ATKIN, M. S., AND WESTBROOK, D. L. Tapir: the evolution of an agent control language. In *Computers and Games 2002* (2002).

[7] KING, G. W., ATKIN, M. S., WESTBROOK, D. L., AND COHEN, P. R. Tapir: an action language beyond scripting. In *The 3rd International Conference on Computers and Games (CG'02)* (2002).

[8] KING, G. W., MORRISON, C. T., WESTBROOK, D. L., AND COHEN, P. R. Bridging the gap: Simulations meet knowledge bases. In *Proceedings of AeroSense 2003* (2003).

[9] KING, G. W., SCHMILL, M., HANNON, A., AND COHEN, P. Asymmetric threat assessment tool (ATAT). In *to appear in Proceedings of the 2005 Behavior Representation in Modeling and Simulation (BRIMS) conference* (2005).

[10] CENTER FOR ARMY LESSONS LEARNED (CALL). A leader's guide to after-action reviews. Tech. rep., Fort Leavenworth, KS, 1993.

[11] DEPARTMENT OF THE ARMY. Field manual 3-90: Tactics. Tech. rep., 2001.

Appendix A – Operational Description of Capture the Flag (as of September 2000)

Research in real-time, adversarial planning has reached the stage that intelligent agents can now play plausibly in simulated war games. At the University of Massachusetts we have developed a wargaming environment called Capture the Flag, and a planner called GRASP which regularly beats human adversaries. This note describes operational features of Capture the Flag – how the system appears and what it provides to operational users. The underlying ideas in planning and simulation are presented in [1, 7, and 8].

War games in Capture the Flag are played in a virtual world of "nearly three dimensions": In addition to latitude and longitude we have elevation, although all hills are the same height. Opposing land and air units attempt to capture each other's flags. They can exploit or be hindered by terrain features such as hills, rivers, and terrain types such as forest and swamp. When units engage, either directly or at a distance, attrition is modeled by modified Lanchester equations. In addition, the dynamics of engagements are influenced by models of psychological factors – fear, morale, fatigue and the like.

Opposing units can be controlled by humans or by the GRASP planner, and a common configuration is for humans to play against GRASP. A natural interface allows human players to direct their units on the battlefield, and the units themselves are capable of intelligent reactive behavior to carry out directives without constant supervision (e.g., one can direct a unit to attack another and it will figure out how best to get there, which formations to adopt, and so on). Tempo strongly influences the outcomes of these games. When human players lose the tempo, GRASP presses its advantage, and a tactical disadvantage quickly spreads to a scenario-wide loss of initiative. The human becomes reactive and eventually loses the game.

How Capture the Flag appears to players

The setup of a Capture the Flag scenario is shown in Figure 1. One panel of the display is a map with units arrayed in a starting configuration. (Configurations are selected by the system but may be changed before the game begins, to model prepositioning.) The other panel shows the parameters of the game. For instance, one can select the number of flags for each side (Red and Blue); the number of brigades of each of six kinds (Armored, Mech., Light, Medium, Airborne and Air Assault); and the number of battalions in each brigade. One can also take shortcuts and add entire divisions of various kinds, then modify their brigade and battalion structure as desired; or, as shown in the left-hand panel of Figure 2, one can specify each scenario unit individually. The other panel in Figure 2 shows a menu to select map views. The most common kind of view presents the military icons, terrain, roads and rivers, control measures, and other "bird's eye view" features of the map, but one can see many other visualizations of the scenario, as we will discuss, shortly.

Floating in the right-hand panel of Figure 1 is the Control Engine menu. It is used to run and stop the simulation and planner; single-step them; take a snapshot so one can replay a game from that situation; restore a situation; and zoom in or out on the map. Generally, humans cannot think fast enough to win a Capture the Flag game in real time, so they usually let the game run for a while and then stop it for a while.

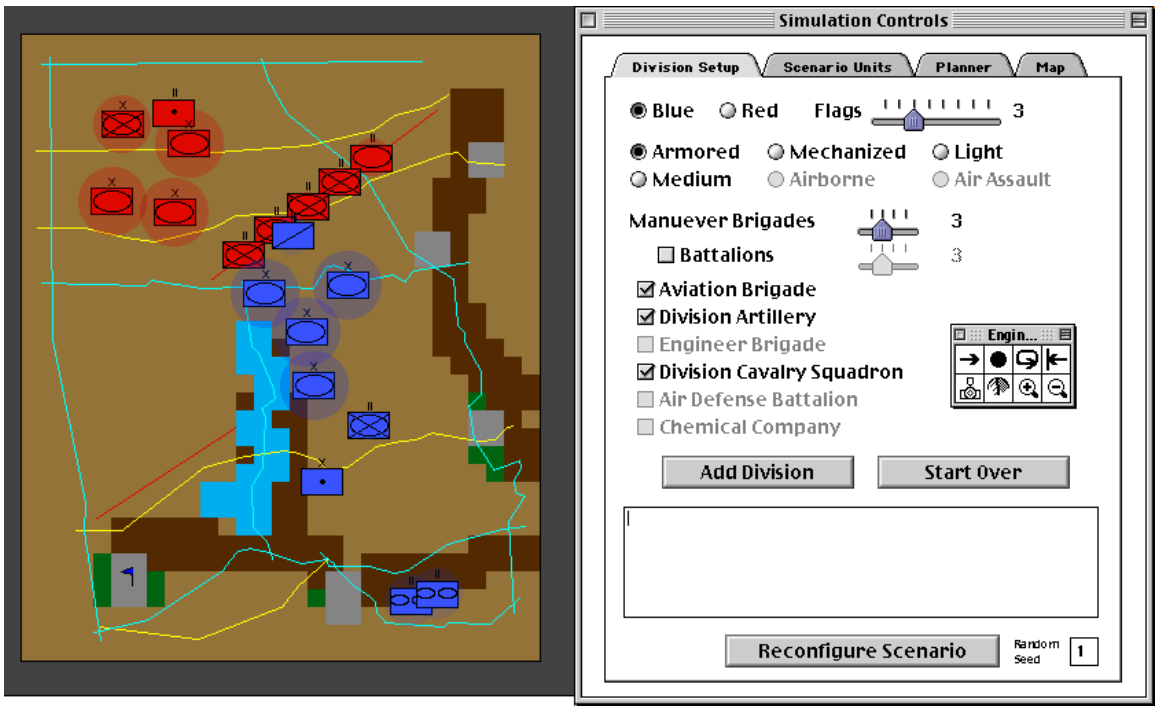

Figure 1. The map of a Capture the Flag scenario, a menu for specifying the parameters of scenarios, and the "control engine" that runs the simulation and planner.

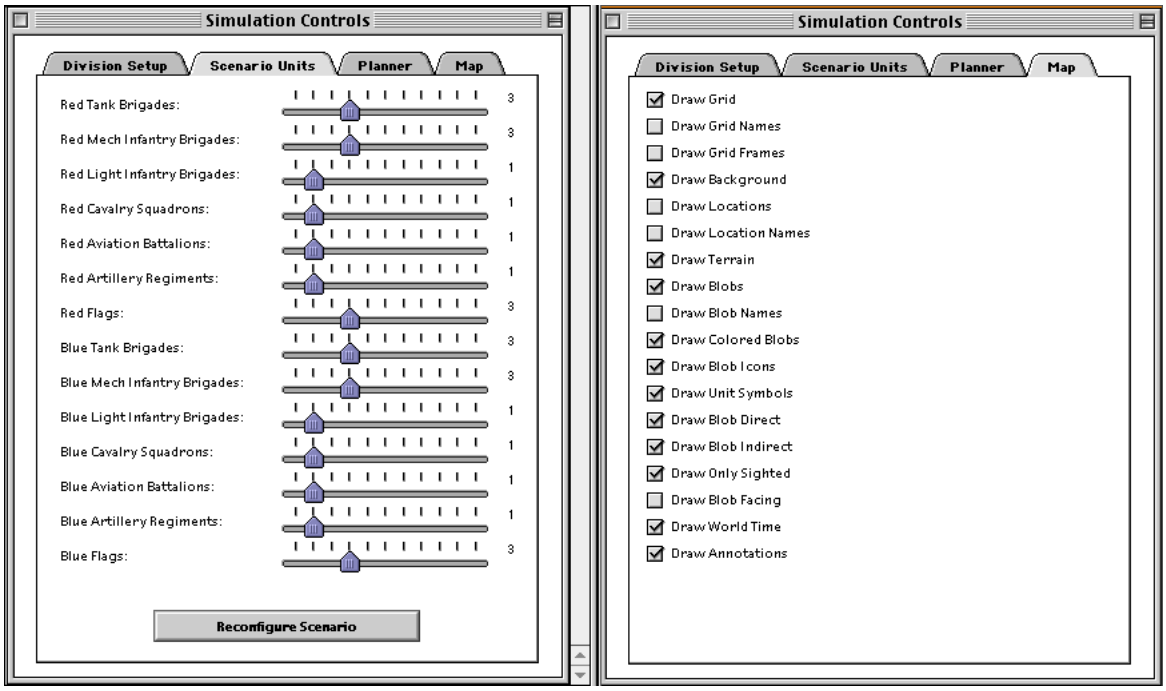

Figure 2. Menus for specifying the units in a Capture the Flag scenario and for configuring the visualizations presented in the map.

The Major Components of Capture the Flag

The Capture the Flag system comprises a simulator (AFS), a control architecture (HAC), and a planner (GRASP). In addition, the system provides GUIs for humans to observe and participate in war games, a variety of visualizations, and instrumentation and other software support to conduct and analyze experiments. One recent GUI is a three-dimensional display shown in Figure 3. When the user clicks on a unit, the display automatically rotates to show the avenues of approach, goals, and threats to the unit.

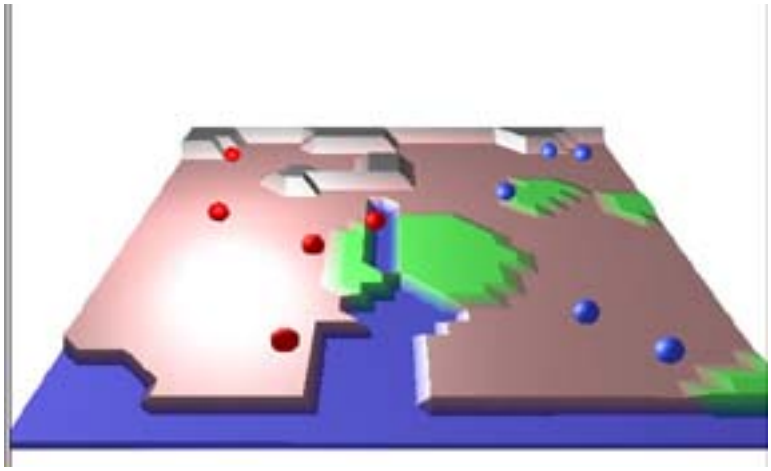

Figure 3. A 3-dimensional visualization. Clicking on a unit causes the display to rotate to an angle that best shows avenues of approach, the chosen trajectory, and targets and threats relevant to the unit.

Playing against the system. A typical experience with Capture the Flag begins with the user designing a scenario, then directing his units (Blue, say) against those of the system. The game plays out in the Abstract Force Simulator (AFS), the dynamical analog of a chessboard. The simulator implements the models that make Capture the Flag realistic; models of movement; terrain; visibility, reconnaissance and sighting; attrition; and mental attributes such as fatigue and morale.

Monte Carlo analysis of COAs. Alternatively, a human may supply a course of action (COA) and direct Capture the Flag to use it to direct, say, Blue units in war games against the GRASP planner. Monte Carlo experiments involve fighting numerous games with roughly the same COA.

When a human player directs a unit to, say, attack another, the Hierarchical Control Architecture (HAC) takes care of the details:

- HAC finds the best route for the unit to follow,
- It avoids obstacles on the route and reroutes units dynamically
- It manages attacks and maneuvers
- It transforms units into different formations as the tactical situation and the terrain require.
- HAC also implements the “eyes and ears” of units, and passes information among units.

The movements of opposing forces (Red, say) are determined by the GRASP planner, which coordinates the efforts of its units, directs them against Blue units and flags, and defends terrain features and flags. Just as human directives are managed by HAC and play out in AFS, the same occurs with GRASP's directives.

Types and Attributes of Units

Capture the Flag provides land and air combat units of the following types: Tank, Mechanized Infantry, Light Infantry, Cavalry, Artillery, and Aviation. The smallest units in Capture the Flag are battalions, the largest, divisions. Unlike MODSAF, for example, our Abstract Force Simulator does not model the movements of individual vehicles, but rather, treats entire units as deformable "blobs." The shapes of blobs change in response to terrain and tactical requirements.

We take Clausewitz's physical metaphors very seriously, so the most primitive attributes of units are physical attribute: mass, density, shape, velocity, resilience, and elasticity. The Abstract Force Simulator (AFS) determines how these attributes change over time in such a way as to model warfare. For example,

- One effect of attrition is to reduce the mass of a unit; attrition can also change the density of the unit (the distribution of its mass within its boundary), or its shape.
- Formations and to some extent maneuvers involve changing shape. Units can adopt columnar and frontal formations, as well as wedge (one up, two back) and "V" (two-up, one back) formations. They can change formation while moving, and while engaged, though in the latter case the process may be slower.
- The velocity of a unit depends on its shape (formation), the kind of terrain it must traverse, what kind of unit it is (e.g., infantry are faster than tank units in wooded terrain), whether it is in combat, and which tactical maneuver it is following (e.g., units move more slowly when they share an avenue of approach).
- When units "dig in," their resilience increases; resilience is resistance to attrition.
- The elasticity of a unit is the propensity for another unit to "bounce off" it. Some tactical situations require units to "absorb" others; for example, a favorite of Sun Tsu's involves absorbing an assault, allowing it to penetrate, then collapsing in on the incoming units.

In addition to these physical attributes of units, we model mental attributes such as morale, fatigue, and uncertainty. Crucially, AFS implements mutual dependencies between physical and mental attributes. Thus, attrition reduces morale, and low morale reduces the ability of a unit to inflict attrition on other units. This particular mutual dependency can lead to a downward spiral in which a unit becomes progressively more damaged, demoralized and unable to fight back.

Visibility. Units have limited sensors, and limited knowledge of the battlefield. A unit becomes visible to another only when air reconnaissance spots it, or when the units make visual contact. Units may hide in woods or behind hills. When humans play against the system, they may select an omniscient view of all units (we call this cheating) or they see only those opposing units that have been sighted.

Tactics and other behaviors

The behaviors of units range from low-level physical adjustments (e.g., changing formation) to movement and bringing fire, to tactical plans such as passage of lines, to emergent behaviors such

as the downward spiral of demoralization mentioned earlier. We have implemented several tactical plans:

- For ground units: *Move* to a location, *occupy* (defend) a location, *retain* a flag, *block*, *follow and assume*, *follow and support*, *forward passage of line*, *direct attack*, *indirect attack* (e.g., by artillery).
- For aviation units, *reconnaissance*, *interdiction*, *close air support*, and establishment of *air superiority*.

We are constantly implementing other tactical plans; those on the agenda include flank attack, envelopment and double envelopment; and hasty and moving defenses.

Many factors affect behavior, and thus the outcomes of tactical plans. For example, suppose unit A attacks unit B, what will happen? It depends on these and other factors:

- The relative masses of A and B (big units tend to win), and the types of the units;
- The terrain on which they engage and the types of the units (e.g., tanks do poorly against infantry in wooded terrain);
- B's degree of entrenchment;
- The time it takes A to move to B (this affects B's entrenchment and A's level of fatigue), which depends on terrain and the type of unit A;
- Whether or not A surprises B (surprise acts as a multiplier on attrition), which itself depends on whether A is visible to B;
- The morale and fatigue levels of A and B
- The formations of A and B

Blobology. Many tactical plans exploit geometry, for example, flank attacks require exposed flanks. Originally, all units in Capture the Flag were circles, and they maintained this shape in all conditions, making it difficult to implement tactical plans. Now, however, units change shape in response to terrain features (e.g., squeezing through a pass), tactical requirements (e.g., moving in column), and combat situations (e.g., a wedge formation may be deformed by contact with the enemy). Some examples are shown in Figure 4.

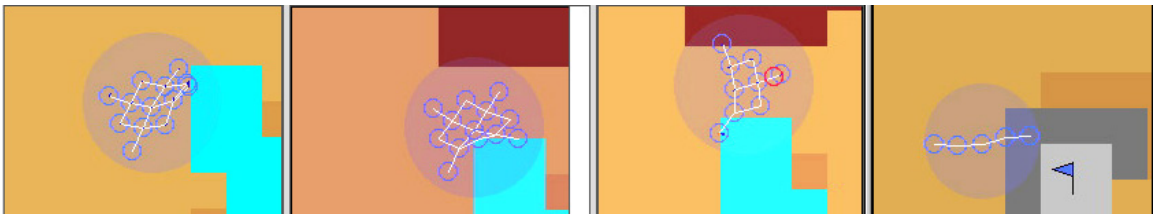

Figure 4. Effects on the morphology of units. From left: A unit compressed by a lake; a unit squeezing through a narrow defile between a lake and an impassable area; a wedge formation moving east, deformed by the lake and impassable area; a unit moving in column.

Appendix B – Original CtF move-to-point code

Though each implementation provides slightly different functionality, the code presented below is roughly equivalent to the Tapir code shown in the section 2.0.1 Tapir: A better action language. This code is196 lines; the Tapir code is only 50.

```
(define-debugging-class move-to-point (move-to-point-actions))

(define-action move-to-point (level-n-action move-to-point-mixin)
  ;; arguments
  ((overcome-block-mode :move-through) ;how should it deal with obstacles
on path (:wait :move-through)
   ;; storage
   (sub-move-failed? nil)
   (avoid? t)              ;try to avoid blobs you encounter?
   (n-sub-moves-failed 0)
   (original-target-geom nil)
   (path-edges nil)
   (est-time-to-complete nil)
   (temp-delay 0)
   (first-call? t))  ;for simulate
  :automatic-accessors
  :automatic-initargs
  :copy-slots
  (:export-p t))

(defparameter *mtp-retries* 1)

(defun mtp-deal-with-failure-from-children (game-state action)

  (incf (n-sub-moves-failed action))

  (if (> (n-sub-moves-failed action) *mtp-retries*)
    (progn
      (when-debugging-format
        move-to-point
        "MTP: *** Switched ~a to 'don't avoid'" action)
      (setf (n-sub-moves-failed action) 0
            (temp-delay action) (uniform-random game-state 5 10))
      (when (eq (overcome-block-mode action) :move-through)
        (setf (avoid? action) nil)))
    ;; else:
    (setf (sub-move-failed? action) t)))

(defmethod handle-message ((game-state game-state) (action move-to-point)
                           (message change-speed))
  (when-debugging-format move-to-point
                         "MTP: (~A) Change speed to ~A / ~A-%."
                         (name (blob action))
                         (speed message)
                         (speed-percent message)))

(defmethod handle-message ((game-state game-state) (action move-to-point)
                           (message all-resources-gone))
  (when-debugging-format move-to-point
                         "MTP: ~a: Received all-resources-gone."
                         action))

(defmethod handle-message ((game-state game-state) (action move-to-point)
                           (message failure))
  (when-debugging-format move-to-point
                         "MTP: ~a: Received failure."
                         action)
```

```
  (mtp-deal-with-failure-from-children game-state action)
  (interrupt-children game-state action))

(defmethod handle-message ((game-state game-state) (action move-to-point)
                           (message no-progress-in-movement))
  (when-debugging-format move-to-point
                         "MTP: ~a: Received no-progress."
                         action)
  (mtp-deal-with-failure-from-children game-state action)
  (interrupt-children game-state action))

(defmethod handle-message ((game-state game-state) (action move-to-point)
                           (message success))
  (unless (original-target-geom action)
    (when-debugging-format move-to-point
                           "MTP: ~a reached ~a!"
                           (blob action)
                           (target-geom action)))
  (complete-action game-state action))

;;?? Do we really need this?
(defmethod check-and-generate-message
           ((game-state game-state) (action move-to-point)
            (type (eql 'all-resources-gone)) (qualifier (eql :BEFORE)))
  (if (not (exists-p (blob action)))
    (progn
      (when-debugging-format move-to-point
                         "MTP: (~a) Blob Died."
                         (name (blob action)))
      (complete-action game-state action 'all-resources-gone)
      (values t))
    (progn
      (values nil))))

(defmethod path-planning-enabled? (game-state blob)
  (if (path-planning-p (scenario game-state))
    (progn
      (afs::ensure-passes-initialized game-state)
      (if (path-profile (path-i (scenario game-state)) (friction-profile
blob))
        (values t)
        (values nil)))
    (values nil)))

 (defmethod realize ((game-state game-state) (action move-to-point))
  (if (> (temp-delay action) 0)
    (progn
      (restart-action game-state action (+ (current-time action)
                                           (temp-delay action)))
      (setf (temp-delay action) 0))
    (let* ((movers (extract-simulatable-members (blob action))))
    (when movers
      (cond ((not (cdr movers))
             (move-to-point-single-blob-internal game-state action))
            (t
             (move-to-point-multiple-blobs-internal    movers    game-state
action)))))))

(defun move-to-point-multiple-blobs-internal (movers game-state action)
  (start-new-child action game-state 'formation-move-to-point
                   :blob (first movers)
                   :followers (rest movers)
                   :target (target-geom action)
                   :resources movers))

(defun move-to-point-single-blob-internal (game-state action)
```

```
  ;; hack to deal with singleton group blobs
  (assert (not (cdr (extract-simulatable-members (blob action))))
          () "should not be a group blob")
  (setf (blob action) (first (extract-simulatable-members (blob action))))

  (let ((avoid-fn (if (eq (overcome-block-mode action) :wait)
                   #'false-fn
                   (if  (avoid?  action)  (blob-avoid-fn  action)  #'false-
fn)))
        (tempy nil))
    (when (eq (overcome-block-mode action) :wait)
      (when (setq tempy
                  (grid-overlap
                   (grid game-state)
                   (make-circle (location-add-vect
                                 (location (blob action))
                                 (vect-mult (velocity (blob action)) 2.0))
                                .5)
                   :test #'(lambda (x)
                             (and (slot-exists-p x 'team)
                                  (eql   (team   x)   (enemy-team   (team
action)))))))
        (when-debugging-format
         move-to-point
         "MPT: Found obstacle ~A! Restarting ~a~%" tempy (blob action))
        (awhen (hac::pending-children action)
          (interrupt-children game-state action))
        (restart-action
         game-state action (+ (current-time game-state) 10))  ;retry again
in 10 ticks
        (return-from move-to-point-single-blob-internal))
      (restart-action game-state action))

    (cond ((path-planning-enabled? game-state (blob action))
           (multiple-value-bind
             (path-edges est-time-to-complete)
             (find-best-path-with-passes game-state (blob action)
                                         (location (blob action))
                                         (location (target-geom action)))
             (setf (path-edges action) path-edges
                   (est-time-to-complete action) est-time-to-complete)

             (when-debugging-format move-to-point
                                    "MTP:  Sending  ~a  to  ~a  via  ~a  (ETA:
~,0f)"
                                    (blob action)
                                    (target-geom action)
                                    path-edges
                                    est-time-to-complete)

             (let* ((waypoints (mapcar
                                (lambda (edge)
                                  (if  (location-equal  (location  (target-
geom action))
                                                        (vertex-loc  (vertex-
to edge)))
                                    (target-geom action)
                                    (make-destination-geom      (vertex-loc
(vertex-to edge)))))
                                   path-edges)))
               ;; Use MWW even for single point paths
               (start-new-child action game-state 'move-with-waypoints
                                :blob (blob action)
                                :resources (extract-blob-resources (blob
action))
                                :waypoint-list waypoints
                                :target-geom (target-geom action)
```

```
                                  :blob-avoid-fn avoid-fn
                                  :speed (speed action)
                                  :speed-percent (speed-percent action)
                                  :messages-to-generate    '(all-resources-
gone completion)))))
             (t
              (start-new-child action game-state 'move-to-point-criteria
                               :blob (blob action)
                               :resources   (extract-blob-resources   (blob
action))
                               :target-geom (target-geom action)
                               :blob-avoid-fn avoid-fn
                               :speed (speed action)
                               :speed-percent (speed-percent action)
                               :messages-to-generate   '(all-resources-gone
completion))))))
```

Appendix C - Tapir

June 17, 2005

1 Overview

Tapir is a semi-declarative action language specification. This section presents the Tapir language at two levels. The first provides a bird's eye view of the language: its main constructs are defined and a brief description of their clauses and options is given. The second returns to each language element and describes its function in detail with numerous examples. To set the rest of the language in context, we begin with a brief overview: Tapir actions are defined with the defaction command. Actions use sensors, resources and child actions to get things done. Resources connect actions to effectors in the world. Particular kinds of resouces are defined with defresource-type. Actual resources in the world are defined with defresource. Sensors connect actions to a representation of the world and are defined with defsensor. Finally, actions often need to commicate with one another and sensors need to communicate with actions. Communication in Tapir occurs via message passing. Messages are defined with defmessage.

2 Key Ideas

2.1 Overview

Actions are connected to the world by sensors and resources. Sensors let them know what is happening. Resources are the effectors that let them act. Actions accomplish their ends either by using resources directly or by starting child actions. This section touches on the highpoints of Tapir's ontology. Following sections then discuss each of these in detail.

2.2 Control

Actions build a control hierarchy by starting children to do their work for them. The hierarchy bottoms up in "primitive" actions that use resources (effectors) to achieve their ends. Actions can create children and structure their control in a multitude of ways. However, there are some control mechanisms that are so typical that they have been canonicalized in actions of their own.

2.3 Resources

Resources are the effectors of an action and come in many types. For example, resources can be serializably reusable, sharable, primitive, compound, consumable and so on. Resources form a hierarchy of their own. For example, a robot is a resource consisting of motor resources, camera resources and so on.

2.4 Sensors

Sensors tie actions to the world. They can be "primitive", connecting to the world more or less directly, or they can be abstract, amalgamating and processing data from multiple sources. Sensors can be shared by multiple actions.

2.5 Messaging

Actions, sensors, and resources communicate via messages. The following are typical message passing scenarios:

- sensors send actions messages when some event in the world occurs.
- child actions send their parent messages when they complete or when something ”unexpected” occurs.
- parents send their children messages when they wish to redirect their activities mid-stream.
- resources send the actions using them messages when the resource is destroyed.

Messages are used in two complementary roles: to pass information and to direct control. As examples, child actions use informational messages to keep their parents informed of their progress (or lack thereof!) and they use control messages to tell their parents that they are completing.

3 Process and Control

Figure 1 displays a schematic of an action's life cycle. In brief: an action is created, enters its do phase (perhaps many times) and then completes. The details of this simplified description are filled in by special initialization and finalization

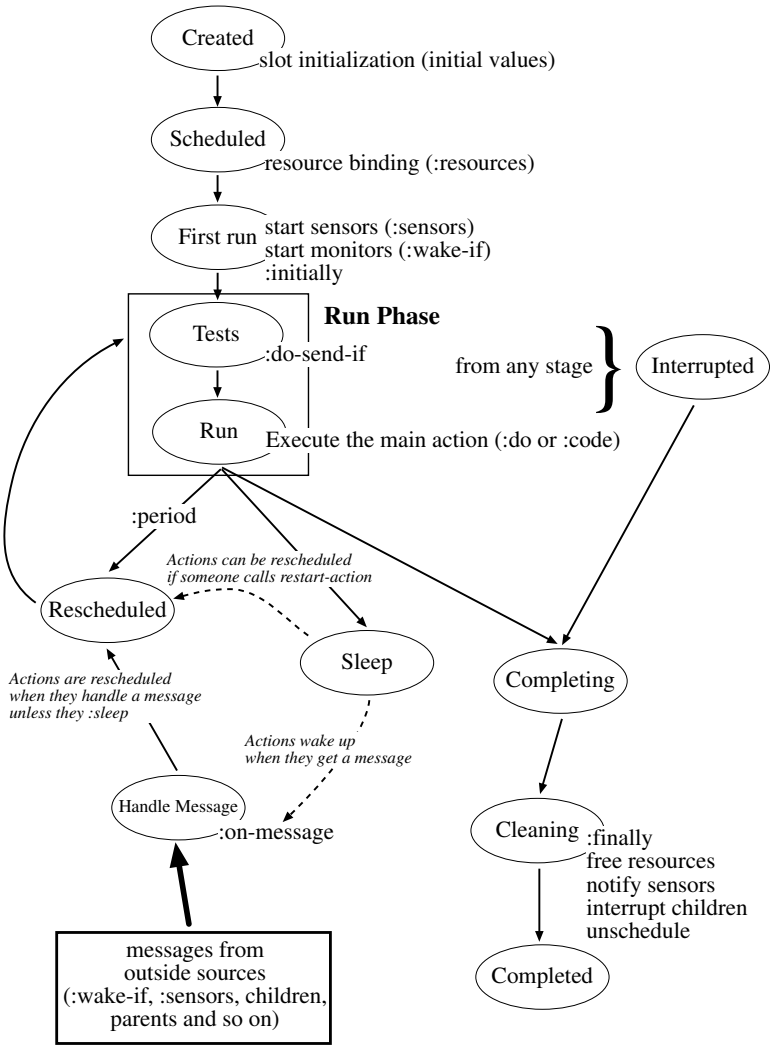

Figure 1: Process Flow

code, sensors, process monitors, child actions, resource utilization and more. Fig-

ure 1 also shows the Tapir language constructs that can be used to influence and describe what an action should do. All of these constructs will be discussed in detail below but first, we will describe the action life cycle in more detail.

Actions maintain internal state in their slots; they also may require resources to do their job. When an action is created, its slots are initialized and any required resources are found and bound. Created actions do not do anything until they are actually scheduled to be run and the time for their activation arrives. When this happens, the action enters its do phase. Actions may execute special code at the beginning of the very first do phase; any sensors required by the action are also run at this time. The do phase itself consists of pre-checks to see if any messages should be sent to the action's parent (these may possibly complete the action without it ever having really executed); the actual execution; and then post-checks to see if any messages should be sent. Once an action executes, it will go to sleep unless it has been rescheduled. A sleeping action remains asleep until it is explicitly rescheduled (perhaps by a parent) or until it receives a message. Actions specify what they would like to do for each different kind of message. Typical responses include: running code, re-entering the do phase, sending the message on up to the action's parent, and ignoring the message completely. When an action is sleeping and it has no sensors or children that can wake it up, the action will automatically complete. Actions can also be interrupted at any time. In both cases, the action enters a cleaning phase where resources are unbound, finalization code is run, children are interrupted and so on.

As mentioned above, messages are used to control the flow of execution between parents and their child actions. Children can complete themselves explicitly by sending a message of type completion (e.g., success or failure) to their parents. If a child completes for some other reason (e.g., a required resource is destroyed) then they will automatically send a completion message during their clean-up phase. Message passing is the only means of control transfer.

4 Seeing Things: defsensor

5 Changing Things: defresource-type and defresource

6 Saying Things: defmessage

7 Doing Things: defaction

Actions in Tapir (and HAC) are classes. Action definitions expand into CLOS class specifications and the set of methods that implement the action. Syntaxically, an action specification looks like:

```
(defAction ([action-superclasses]*)
  clause*
  option*)
```

Action-superclasses are mixins that add specific behavior to an action. DefAction's clauses are summarized here and described in more detail below:

:parameters Describes the parameters of the action. Parameters can be initialized when an action is created. Each parameter becomes a slot in the action's class.

:locals Describes additional local state variables of the action. Locals becomes slots in the action's class.

:documentation An string describing the action.

:resources A specification of the resources required by the action. The resources named in the specification become slots of the action's class.

:sensors Describes an abstract sensor used by the action. Sensors can be named in which case the name of the sensor is added to the slots of the action.

:wake-if Describes monitors for the action and what the action should do when the monitor sends it a message. Monitors can be named in which case the name of the monitor is added to the slots of the action.

:initially Describes what the action should do before it starts. Initially can be code or a child action specification.

:finally Describes what the action should do before it completes.

:code Code that is executed whenever this action runs.

:do A specification of the action's children and how they are to run (e.g., in sequence or repeatedly). Children can be named in which case the name of the child is added to the slots of the action.

:do-send-if-before Describes a test that should occur before the action enters its do phase. It consists of a message specification and code to run. If the parent action requests the message, then the code will be run before each :do phase. If the code evaluates to true, then the message will be sent to the parent.

:do-send-if-after Like :do-send-if-before, only this check is performed after the :do clause runs.

:on-message Describes what the action should do when it receives a message.

:instrument Creates instrumentation to be used by CLIP.

:period Specifies how often this action should be rescheduled.

:debug-name Specifies the name used for the debugging switch associated with this action. See below for more details on debugging.

:debug-superclasses Specifies debugging switches that subsume this action's debug switch. See below for more details on debugging.

:debug Adds debugging code or print statements to the do phase of the action.

7.1 Parameters and Locals

Parameters and locals correspond to slots. Parameters may be passed into an action when it is created. Locals maintain local state. Both specifications may consist of a name (in the form of a symbol) or a list of the form:

```
(name initial-value [{CLOS slot specification} |
  {:read-only | :read-write} |
  {:required | :optional} |
  :export | :copy | :copy-set | :unbound |
  :instrument]*)
```

The initial-value is the value to which the slot is initialized when the action is created. It may be set to :unbound to create actions with slots that are initially not bound to any value. CLOS slot specifications can be used to specify :type, :allocation, :documentation and so on for the slots. The :read-only option creates slots that can only be read. The :required option is only valid for parameters (not locals). If specified, it generates code that ensures that some value is actually supplied for the slot. (Note: in practice, the main difference between parameters and locals is that the former have an :initarg). You can specify that slot names be exported from their package using the :export keyword. You can control the copying behavior of the action's class using the :copy and :copy-set keywords. The :instrument keyword is covered in detail below.

7.2 Message Handling

Here is the new (improved?) proposed syntax for the :handle clause in Tapir. This clause is used to describe how an action should handle the messages that it receives. The clause must be in one of two forms:

```
(:on-message (<type> :ignore)
(:on-message (<type> [code]
                     [(:generate <other-type> args)]
                     [:sleep]
     [:restart])
```

The first type indicates that the message should be ignored. I.e., that no code should be run, that no further message should be sent and that the action should not be run.

The second type is more flexible. It includes optional code (anything that can be in a :code clause, i.e., no action specification), an optional message to generate and the optional :sleep clause. The order does not matter. When a message is received, any code will be run. If there is a :generate clause, a message will be generated. If the :sleep option is present, then the action will not be run. If :sleep is not present, then the action will be run.

7.3 Documentation

The :documentation clause exists to provide a place for people not to use when they fail to write documentation.

7.4 Resources

The resources clause is used to specify what the action needs to accomplish its activities. It consists of a list of names and resource specifications. For example:

```
:resources
 (lw (left-wheel-resource lw))
 (rw (right-wheel-resource rw))
:resources
 (blob (and (indirect-force-capable? blob)
            (> (mass blob) 200)))
```

When the child action is initialized, its parent will hand it the unused resouces. The child will then attempt to match these resources with its resource bindings. If all goes well, the resources will be found and each name will by linked to the specified resource. Resource names become slots in the action's class.

7.5 Sensors

?? Sensor specs look like action specs but start with a sensor.

7.6 Wake-if

The :wake-if clause canonicalizes the standard practice of creating monitors or guardians to ensure that an action is doing what it is supposed to be doing. Wake-if checks a predicate periodically. If the predicate becomes false, any code associated with the wake clause is executed and the action wakes up and enters its run phase.

7.7 Initially

Initialization specifies what should occur the first time that the action is run. You can execute code using the :code keyword and you can start child actions using the :do keyword. You must specify either :do or :code.

7.8 Finally

The :finally clause is meant to help with recovery after something has gone wrong. It may contain any Lisp code.

7.9 Code

The :code clause is code that will be run when the action is scheduled (all else being equal). Actions may have :code or :do but not both.

7.10 Do

The :do clause specifies the potential sub-tasks of an action. It describes both the sub-tasks and how they will be executed. Sub-tasks are the leaves of the :do specification. They can either be Lisp code (using the :code construct) or an action specification consisting of the child action's name and any of its parameters. The sub-tasks can be joined together using the following constructs:

:code Lisp code evaluated as is.

:case Case is Tapir version of Lisp's case statement.

:foreach Dynamically creates a list of children and runs then. The children may be run using any of the child combination types (e.g., in parallel, in sequence and so on).

:generate Though not really a child specification, a :generate clause can be used to send a message to the action's parent from within the :do clause.

:in-parallel Runs a list of children in parallel. Each child in the list is started at the same time and runs independently.

:in-sequence Runs a list of children in sequence. Only one child at a time is active.

:repeat Runs a child action repeatedly for a count or until some predicate becomes true.

:one-of One-of is Tapir's version of a the Lisp cond form. It takes a list of predicates and action specifications. One-of runs the first action specification whose predicate evaluates to true.

:unordered Runs a list of children in some arbitary sequence. Only one child is active at a time but the order of activation is unknown.

:when Each when clause consists of a predicate and an action-specification. If the predicate evaluates to true, the action-specification is run. Multiple when clauses can be used to start several children simultaneously.

Action-specs can nest arbitrarily deep although it is typically more understanable to create complex structure with multiple children rather than building it all into a single action. The following are a few examples:

```
(:do
 (:repeat (:in-sequence
           (:code (let* ((element ...)
                         (complement (dna->rna ...)))
                    (append-element product complement)))
           (move-sub-sequence :sequence sequence))))

(:do
 (:repeat (:in-sequence
           ((add-nucleotide :sequence sequence
                            :product product
                            :offset offset)
            (move-sub-sequence :sequence sequence)))))
```

The syntax and use of children and action specifications are detailed below.

7.11 :do-send-if-before, :do-send-if-after

The :do-send-if clause's are used to send messages only when certain conditions are true. They are evaluated each time an action is scheduled and run. They consist of a message specification and code to evaluate. For example:

```
(:do-send-if-before (success)
  (when (not (null sequence))
    (occupy pool sequence)))
```

In this example, the message specification is simple "success". If the parent action is interested in success messages, then the "(when . . .)" will be evaluated each time the action is about to run. If it returns true (i.e., if the sequence is not null) then the success message will be generated (note that because success is a kind of completion message, the action will complete after it sends the message).

7.12 On-Message

The :on-message clauses describe how an action should react to the messages it receives. Responses can include running code, sending messages to parents and re-running the action.

7.13 Instrument

Instrument specifications are used to build complex CLIPs.

7.14 Debug-name

The debug-name specifies the name used as the debugging switch for this action. If not specified, the debug-name of this action will be the same as the action's name.

7.15 Debug-superclasses

The debug-superclasses clause specifies the names of any debugging switches that should subsume this actions debugging switch. In other words, if debugging superclasses are specified, then they will turn on debugging for this action when they are turned on.

7.16 Debug

This syntax sets the name of the debugging switch for this action. Unless otherwise specified, debugging names are inherited from super-actions. If no debugging name is specified in either the action or its super-actions, then the name of the action itself becomes the debugging name.

8 Debugging

Many clauses can contain a :debug sub-clause. This should look like one of the following:

```
:debug [debug-switch-name] format-string [arguments*]
:debug [debug-switch-name] form*
```

In the former case, the format-string and arguments will be used to print a message when this action has debugging turned on. In the later, the forms will be executed. In both cases, all of the slots of the action will be available. Clauses may have multiple :debug clauses which will be evaluated in the order that they are found. The following clauses can include the :debug switch: :initially, :finally, :wake-if, :on-message, :code, :do, :do-send-if-before, :do-send-if-after. If unspecified, the optional debug-switch-name defaults to the debug-name of the action. You can use this to specify alternate debug switches for more complex debugging behavior.

9 Do: Get someone else to do the work

An action uses its :do clause to specify its subtasks and how these subtasks should be run. Subtasks are either Lisp code (embedded in a :code clause) or task specifications. A task specification is a list consisting of the name of the task and any arguments that are needed. For example:

```
(:code (setf next-location (find-next-location)))

(move-to-point :blob self
               :target-geom destination-circle)
```

Task specifications can be arranged and sequenced in a wide variety of ways. They can run sequentially or in parallel; they can run only under certain conditions and then can be run once or multiple times.

9.1 :case - choose an action to run depending on some value

The :case clause has the following syntax:

```
(:case value-form
 (key action-spec)*
 [(:otherwise action-spec)])
```

During action execution, the value-form is evaluated. Then each of the (key action-spec) forms are examined. If the value of the value-form is the same as that of the key, the action-spec is executed. If the value matches no keys and there is an :otherwise form, then the :otherwise form's action-spec is executed. If there is no :otherwise clause, then no action-spec is executed.

9.2 :code

Any code may be included in a :code section.

9.3 :debug

The :debug clause makes it easy to insert debugging code and messages into the context of an action specification.

9.4 :foreach - do the same thing different ways

Sometimes the number of children is not known until run time. For example, a swarm action will not know how many blobs to swarm until it actually runs and a move-with-waypoints action will not know how many waypoints it will move through until the actual path is planned. The :foreach clause exists to handle these cases. Its syntax is

```
(:foreach variable [in|:in] list mode [mode-specifier]
  action-spec)

(:foreach binding-specification :in list mode [mode-specifier]
  action-spec)
```

The second form is a generalization of the first. In either case, the list is evaluated at run-time. Then the action sets the variable to each item in the list and creates an action based on the action-spec and the current value of the variable. Finally, these actions are :executed either sequentially, in-parallel or unordered, depending on the mode. Here are some examples.

```
;; swarm
(defaction move-em-all-around ()
 (:resources ((blobs t :count :all)))
 (:do
  (:foreach b in blobs :in-parallel
    (move-to-random-point :blob b)))
 :ignore-messages)

 ;; a simple move with waypoints action
 (defaction move-with-waypoints ()
```

```
      (:parameters (waypoint-list))
      (:do
       (:foreach point in waypoint-list :in-sequence
        :until-one-fails
        (move-to-location :blob blob
                  :location point))))
```

9.5 :in-parallel - all together now

When several actions need to be run at the same time (i.e., as separate processes), the :in-parallel clause is indispensable. It has the following syntax:

```
    (:in-parallel [mode]
      action-spec*)
```

The optional mode specifies how to deal with completion. It can be one of :until-one-completes or :until-all-complete (the default). When :until-one-completes is used, the entire set of parallel actions will be interrupted as soon as one of them completes and the parent action will then continue its own processing. If :until-all-complete is specified, the parent will not be notified until all of the parallel actions are finished. Here is how it looks in a simplified envelopment action:

```
     (defaction simple-envelopment ()
      (:parameters ((target-blob :required)
    (flank :right-side)))
      (:resources ((holder
    (and (direct-fire-capable? holder)
 (> (mass holder) (mass target-blob))))
   (mover
    (and (direct-fire-capable? mover)
 (> (mass holder) (/ (mass target-blob) 3.0))))))
      (:do
(:in-parallel :until-one-completes
 (fix :blob holder
      :target-blob target-blob)
 (flank :blob mover
:target-blob target-blob))))
```

Envelopment consists of one blob holding (fixing) the target in place while another attempts to flank it. The fixing and the flanking happen at the same time and the envelopment should end when either of its children completes.

9.6 :in-sequence - line up and wait your turn

The :in-sequence specifier runs several actions one after the other in order. Its syntax is similar to the :in-parallel clause:

```
(:in-sequence [mode]
  action-spec*)
```

The mode argument can be :until-one-fails or :until-one-succeeds. The first will run each action in the specification until one of them fails; the second will run each action until one of them succeeds. For example, suppose there are several ways to accomplish a goal. One could use :in-sequence :until-one-succeeds to try each method sequentially and stop as soon as one of them succeeded. On the other hand, if some multistep process can only be achieved if every step works, then :until-one-fails will prevent the action from continuing on to a fruitless end

9.7 :one-of - keep trying until one fits

```
(:one-of (predicate action-spec)*
         [(:otherwise action-spec))
```

9.8 :repeat - one good turn deserves another

```
(:repeat [count]
  [:when predicate]
  [:unless predicate]
  [:times {expression | number}]
  action-spec*)
```

9.9 :unordered - one at a time but who knows which?

An :unordered action runs each of its steps exactly as :in-sequence except that the ordering of the steps is not known (it is randomly choosen at run-time).

9.10 :when - do this on condition

```
(:when predicate
  action-spec*)
```

10 Instrumentation

Tapir supports writing instrumentation for actions using EKSL's CLIP package. Clips are functions that return some useful value of a software system. For example, clips for an action might include the number of times it was run, the value of some slot of the action, how often the action succeeds, and so forth. Clips are assembled into experiments: the CLIP package will record the output values of clips when varying some experimental variables which you specify. The same clip can be used in many experiments. Experiments generate data that can be used in any standard statistical analysis package, such as CLASP.

Tapir contains directives for automatically generating clips for actions if you say when the clip should run and what it should output. In order to generate data, the user must tell CLIP how to run the experiment using the `define-simulator` and `define-experiment` macros (see the CLIP manual). Alternatively, you can use `DEFINE-TAPIR-EXPERIMENT` (q.v.).

In addition, you can write clips directly with `CLIP:DEFCLIP` and freely use them in the same experiments as Tapir-defined clips. This allows you to use Tapir to write clips that are action-specific, and to use CLIP directly to write clips that apply more globally.

In general, there are two kinds of clips: simple clips and time-series clips. Simple clips are called only once, when the data file is generated, and generate only one row of data. Time-series clips generate mulitple rows of data in the output file whenever some condition occurs. If you define a clip using Tapir, we presume that you'd like to know the clip's value whenever an instance of that action runs. Therefore, every clip generated by Tapir is a time-series clip.

Clips can be defined using Tapir in two ways: during a slot definition, or using the `:instrument` Tapir directive. Within a slot definition using `:parameters` or `:locals`, you can add the flag `:instrument` to automatically define a time-series clip which runs when the action completes and returns the value of that slot. To change when the clip runs or other clip options, you can specify the flag like (`:instrument` *options*), where *options* can be any options allowable for the `:instrument` directive.

Here is how the `:instrument` directive works.

Tapir directive `:instrument` (*name options body*)

Defines a clip for this action.

The list *options* may be any options allowable for `CLIP:DEFCLIP`, with the following additions:

- (`:columns` column-list) States that the values returned by the clip should be named *column-list*. This is exactly equivalent to the `:components` option in CLIP.
- (`:trigger` <:start | :complete | :message>) Specifies when the clip should output its data. :start means when the action starts, :complete means when the action finishes, and :message means whenever the action receives a message.
- (`:period` <n>) Specifies when the clip should output its data. This means that the clip should output every n-th time the action's realize method is run. Of course, n must be a positive integer. Only one of `:period` and `:trigger` should be specified.

The body of the instrumentation, as in CLIP, is simply arbitrary Lisp code that should return one value for each experimental variable that you want the clip to return. Within the body of instrumentation, similarly to in a `:DO` directive, the variable `ACTION` is bound to the current action object. The variable `THE-SIMULATION` is bound to the current game-state, i.e., the datastructure that contains all of the simulation state.

Function `ACTION-INSTRUMENTATION` (*action*)

Returns a list of symbols that denote all the clips defined by Tapir for ACTION. Clips defined outside of Tapir—i.e., using `CLIP:DEFCLIP` directly—will not be included.

Macro `DEFINE-TAPIR-EXPERIMENT` (*name args options*)

Defines an experiment for use with CLIP. This macro has exactly the same syntax as CLIP:DEFINE-EXPERIMENT (in fact, it expands into a call to DEFINE-EXPERIMENT) except that it accepts one additional option.

- `:actions` ACTION-LIST Adds all the clips defined by Tapir for ACTION-LIST to the list of instrumentation for the experiment. This may be freely mixed with the `:instrumentation` keyword.